38th International Symposium on Distributed Computing

DISC 2024, October 28–November 1, 2024, Madrid, Spain

Edited by

Dan Alistarh

Editors

Dan Alistarh
Institute of Science and Technology Austria (ISTA), 3400 Klosterneuburg, Austria
dan.alistarh@ist.ac.at

ACM Classification 2012
Software and its engineering → Distributed systems organizing principles; Computing methodologies → Distributed computing methodologies; Computing methodologies → Concurrent computing methodologies; Hardware → Fault tolerance; Information systems → Data structures; Networks; Theory of computation; Theory of computation → Models of computation; Theory of computation → Design and analysis of algorithms

ISBN 978-3-95977-352-2

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-352-2.

Publication date
October, 2024

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.DISC.2024.0

ISBN 978-3-95977-352-2 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Regular Papers

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Brief Announcements

Preface

DISC 2024, the 38th International Symposium on Distributed Computing, was held between October 28th and November 1st, 2024, in Madrid, Spain. DISC is an international forum on the theory, design, analysis, and implementation of distributed systems and networks, focusing on distributed computing. DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS).

Statistics

DISC 2024 received 170 submissions in the "regular paper" category, and 9 submissions in the "brief announcement" category. The program was selected by a program committee consisting of 37 full members and 2 half-load members. The program committee was assisted by 111 external reviewers. As usual for DISC, the committee used a relaxed form of double-blind reviewing, where the submissions themselves were anonymous, but authors were permitted to disseminate their work by uploading it to online repositories or by giving talks about it. Each submission was evaluated by at least three reviewers, and final decisions were made during a 2-day virtual PC meeting, during which approximately 30 submissions were discussed.

The final statistics are as follows:

- 39 submissions were accepted as regular papers, for an acceptance rate of $\sim 23\%$;
- 16 brief announcements were accepted, of which two were submitted in this form, and 14 are short versions of full paper submissions.

The keynote talks at DISC 2024 were given by Stephanie Gil (Harvard University), Stefan Schmid (TU Berlin), and by Gauri Joshi (Carnegie Mellon University).

Awards

The following two awards are jointly sponsored by DISC and the ACM Symposium on Principles of Distributed Computing (PODC):

- The **2024 Edsger W. Dijkstra Prize in Distributed Computing** was awarded to Nicola Santoro and Peter Widmayer for their paper: "Time is Not a Healer" which originally appeared in the Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS), pages 304–313, 1989. The paper introduced the fundamental notion of dynamic transmission faults, with the goal of modeling message losses on a communication channel, in an otherwise synchronous system. As such, it was the first to investigate the impact of a changing communication topology during the execution of the algorithm on the solvability of distributed agreement tasks, enriching our understanding of this area, and leading to significant follow-up work. The prize was awarded to the authors at PODC 2024 in Nantes.
- The **2024 Principles of Distributed Computing Doctoral Dissertation Award** was presented at DISC 2024. The committee decided to share the award between two recipients: Dr. Robin Vacus for his dissertation "Algorithmic Perspectives to Collective Natural Phenomena," and Dr. Yaunhao Wei for his dissertation "General Techniques for Efficient Concurrent Data Structures."

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This volume also includes the citations the best paper and best student paper awards at DISC 2024, as well as the citations for the 2024 Edsger W. Dijkstra Prize in Distributed Computing, which was presented at PODC 2024, and for the Best Dissertation Awards, which were presented at DISC 2024.

Acknowledgments

I would like to end by wholeheartedly thanking everyone who contributed to this edition of DISC: the authors who submitted their work to DISC, the PC members and external reviewers who helped formed the DISC 2024 program, the keynote speakers, the organizing committee, and in particular the local chair Alexey Gotsman, the workshop chairs, and the members of the award committees. Further, I would like to thank Joel Rybicki for help with the proceedings, John Lazarsfeld for help running the program committee meeting, and William Moses, Jr. for handling DISC 2024 publicity.

I am also extremely grateful to all the members of the steering committee, who supported me significantly throughout the process, and to former chairs of DISC, in particular Rotem Oshman, who provided extremely useful practical information and advice. Finally, I would like to thank EATCS for their support, and the staff Schloss Dagstuhl – Leibniz-Zentrum für Informatik for their help in preparing these proceedings.

November 2024

Dan Alistarh
DISC 2024 Program Chair

Organization

The International Symposium on Distributed Computing (DISC), is an annual forum for the presentation of research on all aspects of distributed computing. It is organized in cooperation with the European Association for Theoretical Computer Science (EATCS). The symposium was established in 1985 as a biannual International Workshop on Distributed Algorithms on Graphs (WDAG). The scope was soon extended to cover all aspects of distributed algorithms and WDAG came to stand for International Workshop on Distributed AlGorithms, becoming an annual symposium in 1989. To reflect the expansion of its area of interest, the name was changed to DISC (International Symposium on DIStributed Computing) in 1998, opening the symposium to all aspects of distributed computing. The aim of DISC is to reflect the exciting and rapid developments in this field.

Program Chair

Dan Alistarh	Institute of Science and Technology (ISTA), Austria

Program Committee

Ittai Abraham	Intel (Israel)
Vitaly Aksenov	City University of London (United Kingdom)
Dan Alistarh	ISTA (Austria), Chair
Hagit Attiya	Technion (Israel)
Georgia Avarikioti	TU Vienna (Austria)
Naama Ben-David	Technion (Israel)
Janna Burman	Paris-Saclay University, LISN (France)
Armando Castañeda	Universidad Nacional Autónoma de México (UNAM) (Mexico)
Yi-Jun Chang	National University of Singapore (Singapore)
Bogdan Chlebus	Augusta University (USA)
Shir Cohen	Cornell (USA)
Varsha Dani	Rochester Institute of Technology (USA)
Peter Davies-Peck	Durham University (United Kingdom)
Laxman Dhulipala	University of Maryland and Google (USA)
Michal Dory	University of Haifa (Israel)
Faith Ellen	University of Toronto (Canada)
Manuela Fischer	ETH Zurich (Switzerland)
George Giakkoupis	INRIA Rennes (France)
Seth Gilbert	National University of Singapore (Singapore)
Alexey Gotsman	IMDEA (Spain)
Siddhartha Visveswara Jayanti	Google Research (USA)
Mohsen Lesani	University of California, Santa Cruz (USA)
Julian Loss	CISPA Helmholtz Center for Information Security (Germany)
Yannic Maus	TU Graz (Austria)
Darya Melnyk	TU Berlin (Germany)
Alessia Milani	Aix-Marseille University LIS (France)
Giorgi Nadiradze	Aptos (USA)

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Dennis Olivetti	Gran Sasso Science Institute (Italy)
Rotem Oshman	Tel-Aviv University and Princeton University (Israel/USA)
Peter Robinson	Augusta University (USA)
Joel Rybicki	HU Berlin (Germany)
Jared Saia	University of New Mexico (USA)
Thomas Sauerwald	Cambridge University (United Kingdom)
Christian Scheideler	Paderborn University (Germany)
Stefan Schmid	TU Berlin (Germany)
Lili Su	Northeastern University (USA)
Jukka Suomela	Aalto University (Finland)
Marko Vukolic	Protocol Labs (Switzerland)
Leqi Zhu	ISTA (Austria)

Organizing Committee

Alexey Gotsman	IMDEA Software Institute (Spain)
Antonio Fernández Anta	IMDEA Networks Institute (Spain)
Yannic Maus	TU Graz (Austria), Workshops Chair
William K. Moses Jr.	Durham University (United Kingdom), Publicity Chair
Joel Rybicki	HU Berlin (Germany), Proceedings Chair

Steering Committee

Jukka Suomela	Aalto University (Finland), Chair
Hagit Attiya	Technion (Israel), Vice-Chair
Christian Scheideler	University of Paderborn (Germany)
Rotem Oshman	Tel Aviv University (Israel)
Seth Gilbert	NUS (Singapore)
Yannic Maus	TU Graz (Austria)

External Reviewers

Sharareh Alipour
James Aspnes
John Augustine
Lukas Aumayer
Alkida Balliu
Joffroy Beauquier
Trevor Brown
Costas Busch
Christian Cachin
Benjamin Chan
Chaodong Cheng
Arka Rai Choudhuri
Daniel Collins
Emilio Cruciani
Poulami Das
Shantanu Das
Quinten De Man
Dipan Dey
Fabien Dufoulon
Antoine El-Hayek
Robert Elsässer
Yuval Emek
Constantin Enea
Ali Farahbakhsh
Alexander Fedorov
Nick Fischer
Orr Fischer
Maxime Flin
Pierre Fraigniaud
Matthias Függer
Hugo Rincon Galeana
Rati Gelashvili
Anna Geisler
Yuval Gil
Jeff Giliberto
Wojciech Golab
Themis Gouleakis
Magnus M. Halldorsson
Thomas P. Hayes
Juho Hirvonen
Gary Hoppenworth
Xing Hu

Shang-En Huang
David Ilcinkas
Sucharita Jayanty
Valerie King
Lucianna Kiffer
Peter Kling
Ajay Kshemkalyani
Ralf Kuestera
Petr Kuznetsov
Arnaud Labourel
Rustam Latypov
John Lazarsfeld
John Lazarsfeld sysadmin
François Le Gall
Christoph Lenzen
Peter Li
Dimitrios Los
Giuliano Losa
Akaki Mamageiashvili
Elad Michael Schiller
Gopinath Mishrai
Slobodan Mitrović
Masayuki Miyamoto
Kaushik Mondal
William K. Moses Jr.
Yoram Moses
Raïssa Nataf
Yasamin Nazari
Ray Neiheiser
Calvin Newport
Thomas Nowak
Tsutomu Okano
Charlotte Out
Shreyas Pai
Sergei Pankratov
Ami Paz
Sriram V. Pemmaraju
Matthieu Perrin
Seth Pettie
Rafael Pinot
Nuno Preguica
Mikaël Rabie
Sergio Rajsbaum
Michel Raynal
Wen Richard
Nicolas Rivera
Atri Rudra
Mher Safaryan
Giulia Scafino
Noa Schiller
Gal Sela
Gokarna Sharma
Kecheng Shi
Gilad Stern
Hsin-Hao Su
Yuichi Sudo
Pierre Sutra
John Sylvester
Gadi Taubenfeld
Amitabh Trehan
Jara Uitto
Manuel Vidigueira
Hoa Vu
Koichi Wada
Haochen Wang
Yuanhao Wei
Jennifer Welch
Julian Werthmann
Luca Zanetti

Acknowledgements

DISC 2024 was organized by the IMDEA Software Institute. The travel of the invited speakers to DISC was supported by grant CNS2023-144164, funded by MCIN/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/PRTR.

DISC 2024 acknowledges the use of HotCRP for handling submissions and managing the review process and LIPIcs for producing and publishing the proceedings.

DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS).

Distinguished Paper Awards

Best Papers

The DISC Program Committee has selected the following two papers to share the DISC 2024 best paper award:

Hyperproperty-Preserving Register Specifications
by Yoav Ben Shimon, Ori Lahav, and Sharon Shoham.

Reasoning about hyperproperties of concurrent implementations, such as the guarantees these implementations provide to “client” programs, has been a central area in distributed computing. This paper makes significant contributions to this area by introducing novel concepts such as “complete” implementations and “decisive linearizability.” The authors provide a comprehensive framework for understanding and analyzing the preservation of hyperproperties in shared object implementations, extending beyond traditional linearizability. This research opens up new avenues for simplifying reasoning about concurrent systems and their complex behaviors. The paper’s clear presentation, technical depth, and potential for far-reaching impact in both theory and practice make it a standout contribution to this year’s program. This work also receives this year’s Best Student Paper award.

Lock-Free Augmented Trees
by Panagiota Fatourou and Eric Ruppert.

The paper introduces an elegant and efficient method for maintaining aggregate information in concurrent tree data structures, addressing a critical challenge in parallel computing. The authors’ ingenious propagation technique enables the augmentation of both static and dynamic tree structures with powerful query capabilities while preserving lock-free concurrency and linearizability. The work demonstrates the technique’s applicability to tries and binary search trees, rigorously proving its correctness and efficiency. The work could have significant impact on both theoretical and practical aspects of concurrent programming.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2024 Principles of Distributed Computing Doctoral Dissertation Awards

The committee for the 2024 Principles of Distributed Computing Doctoral Dissertation Award decided to share the award between two recipients:

- Dr. Robin Vacus for his dissertation “Algorithmic Perspectives to Collective Natural Phenomena.”
- Dr. Yaunhao Wei for his dissertation “General Techniques for Efficient Concurrent Data Structures.”

Dr. Vacus’s Dissertation

Dr. Vacus completed bis PhD under the supervision of Amos Korman and Pierre Fraigniaud, at the Université Paris Cité. His thesis applies a distributed systems approach to problems and models inspired by biology and sociology. The first part of the thesis considers solutions to two agreement-related problems in a setting in which agents have very limited resources, as one would expect in an algorithm that may be executed by animals or even biological cells. It starts by studying a “bit dissemination” problem in which the agents need to decide among two alternatives. Each starts with an opinion but only one of the agents knows the correct choice and will insist on it. Agents exchange opinions with a small sample of peers. The analysis shows an exponential gap beween convergence times in the case in which agents move simultaneously vs. moving sequentially, and a similar gap between memoryless solutions and ones that employ strong separation between the simultaneous and the sequential activation models, and between memory-less solutions and ones in which agents use a small amount of memory. The next problem tacked in this part involves a continuous setting, in which agents try to come as close to their center of mass as possible, while they suffer from Gaussian drift over time and from noisy distance measurements. Somewhat unexpectedly, it is shown that an algorithm using all-to-all communication is not significantly better than one that employs no communication whatsoever. The second part of the thesis considers the role and impact of altruism vs free riding on cooperation in a game-theoretic setting. In one game, it is shown that players’ motivation to work to increase their payoffs can sometimes be positively be affected by the amount of easily accessible resources (“low hanging fruit”), while in other cases it may be negatively correlated to that amount. The final question studied in the thesis is a variant of the “tragedy-of-the-commons” in which besides cooperating or defecting players may opt to behave hypocritically, meaning that they perform the least amount of work needed in order to appear to be cooperating. An original mechanism that uses moderate social pressure on non-cooperators is shown to cause defectors to be more cooperative. Dr. Vacus’ thesis provides an inspiring overview of the questions studied, and employs a wide range of tools and techniques, involving probabilistic analysis, control theory, statistics and game theory, and computer simulations.

Dr. Wei’s Dissertation

Dr. Wei completed his PhD under the supervision of Prof. Guy E. Blelloch, at CMU. In his thesis, Dr. Wei proposes general techniques for improving existing concurrent data structures by simplifying their design and enhancing their performance. The goal of the thesis is to offer techniques that are easy to use to non-experts, even though their implementation

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

behind-the-scenes is complicated and subtle. The techniques presented in the thesis are: (a) Lock-free locks, an automated and general method for converting lock-based concurrent code into lock-free code, requiring no involvement from the programmer; (b) Consistent snapshots: a method for enriching any data structure with a linearizable snapshot operation, which provides a global copy of the state of the object as it existed at some point during the snapshot operation; and (c) Safe memory reclamation: a combination of manual safe-memory reclamation and automated reference counting, which is simpler than existing techniques, and is shown to be competitive in its performance. The thesis also includes implementations and a rigorous empirical evaluation of the techniques it contributes, including applications to a variety of concurrent data structures. The implementations are offered as libraries which are freely available to the public. Given the growing importance of concurrency, and the well-known difficulty of writing correct and efficient concurrent code, the thesis is well-poised to find practical impact in the programming world.

The 2024 Principles of Distributed Computing Doctoral Dissertation Award Committee:
Magnús M. Halldórsson, Reykjavik University
Yoram Moses (chair), Technion
Rotem Oshman, Tel-Aviv University
Paul Spirakis, University of Liverpool and University of Patras

2024 Edsger W. Dijkstra Prize in Distributed Computing

The Edsger W. Dijkstra Prize in Distributed Computing is awarded for outstanding papers on the principles of distributed computing, whose significance and impact on the theory or practice of distributed computing have been evident for at least a decade. It is sponsored jointly by the ACM Symposium on Principles of Distributed Computing (PODC) and the EATCS Symposium on Distributed Computing (DISC). The prize is presented annually, with the presentation taking place alternately at PODC and DISC.

The committee decided to award the 2024 Edsger W. Dijkstra Prize in Distributed Computing to **Nicola Santoro** and **Peter Widmayer** for their paper:

"Time is Not a Healer"
appearing in
Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science,
pages 304–313, 1989.

The paper introduced the fundamental notion of dynamic transmission faults, with the goal of modeling message losses on a communication channel, in an otherwise synchronous system. As such, it was the first to investigate the impact of a changing communication topology during the execution of the algorithm on the solvability of distributed agreement tasks, complementing the classic processor crash fault model.

Beyond this modeling contribution, the paper also showed, via an elegant proof, the surprising technical fact that, in a system with sufficiently many dynamic transmission faults, a weak version of the Consensus problem is "either trivial or impossible." More precisely, Consensus is unsolvable in a synchronous system if an adversary is able to cause up to $n-1$ messages to be lost in every communication round. This illustrated, for the first time, that the impossibility of Consensus can be also caused by insufficient communication, rather than just the lack of synchrony.

These insights have been very impactful over time, highlighting the connection between the communication topology and the computational power of a distributed system. In turn, the paper has had broad influence across diverse areas such as fault-tolerance, agreement problems, dynamic communication networks, and even topological understanding of distributed computing. The paper has also become a classic text thanks to its excellent exposition.

In summary, the seminal paper by Santoro and Widmayer combines original conceptual contributions with deep theoretical insights, and stands out as a significant stepping stone in our theoretical understanding of distributed computing.

The 2024 Award Committee:

Dan Alistarh (chair), ISTA
Shlomi Dolev, Ben-Gurion University of the Negev
Faith Ellen, University of Toronto
Fabian Kuhn, University of Freiburg
Petr Kuznetsov, Telecom Paris & Institut Polytechnique Paris
Jukka Suomela, Aalto University

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Fully Local Succinct Distributed Arguments

Eden Aldema Tshuva
Tel Aviv University, Israel

Rotem Oshman
Tel Aviv University, Israel

Abstract

Distributed certification is a proof system for detecting illegal network states or improper execution of distributed algorithms. A certification scheme consists of a proving algorithm, which assigns a certificate to each node, and a verification algorithm where nodes use these certificates to decide whether to accept or reject. The system must ensure that all nodes accept if and only if the network is in a legal state, adhering to the principles of completeness and soundness. The main goal is to design a scheme where the verification process is local and the certificates are succinct, while using as efficient as possible proving algorithm.

In cryptographic proof systems, the soundness requirement is often relaxed to computational soundness, where soundness is guaranteed only against computationally bounded adversaries. Computationally sound proof systems are called arguments. Recently, Aldema Tshuva, Boyle, Cohen, Moran, and Oshman (TCC 2023) showed that succinct distributed arguments can be used to enable any polynomially bounded distributed algorithm to certify its execution with polylogarithmic-length certificates. However, their approach required a global communication phase, adding $O(D)$ communication rounds in networks of diameter D, which limits its applicability to local algorithms.

In this work, we give the first construction of a fully local succinct distributed argument system, where the prover and the verifier are both local. We show that a distributed algorithm that runs in R rounds, has polynomial local computation, and messages of B bits each can be compiled into a self-certifying algorithm that runs in $R + \text{polylog(n)}$ rounds and sends messages of size $B + \text{polylog(n)}$, with certificates of length $\text{polylog}(n)$. This construction has several applications, including self-certification for local algorithms, ongoing certification of long-lived algorithms, and efficient local mending of the certificates when the network changes.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols

Keywords and phrases distributed certification, proof labeling schemes, SNARG

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.1

Funding *Eden Aldema Tshuva*: Supported in part by AFOSR Award FA9550-23-1-0312, and an Algorand Foundation grant.
Rotem Oshman: Research funded by the Israel Science Foundation, Grant No. 2801/20, and also supported by Len Blavatnik and the Blavatnik Family Foundation.

1 Introduction

In this work we study *distributed certification*, a mechanism that is useful for ensuring correctness and fault-tolerance in distributed algorithms: the goal is to efficiently check, on demand, whether the system is in a legal state or not. To that end, the network computes in advance auxiliary information in the form of *certificates* stored at the nodes of the network, and we design an efficient *verification procedure* that allows the nodes to interact with one another and use their certificates to verify that the system is in a legal state. Since we do not trust that the system is in a legal state at verification time, we think of the certificates as being provided by an *untrusted prover*, whose goal is to convince us that the system is in a legal state even when it is not. One can therefore view distributed certification as a distributed analog of NP.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 1; pp. 1:1–1:24
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Distributed certification was implicit in early work on fault detection and self-stabilization (e.g., [4]), as a mechanism for detecting that the network has changed (for instance, due to the failure of a communication link) and action must be taken to address the change. It was formalized as an object of independent interest in [34], and has since received significant attention in distributed computing literature (e.g., [32, 33, 12, 20, 25, 22, 17, 42, 40, 23, 22, 30, 41, 9]). Almost all work in the area is solely concerned with optimizing the length of the certificates, which is viewed as a proxy for the efficiency of the verification algorithm: in [34] and most of the follow-up work, the verification algorithm consists of a single round of communication, where nodes send their certificates to their neighbors, and then output a local decision whether to accept or reject. Our work departs from most of the literature on distributed certification in two important ways: first, in addition to the certificate length, we are also concerned with the efficiency of the *prover algorithm*, that is, the algorithm that computes the certificates; and second, following [2], we relax the correctness requirement from *perfect soundness* to *computational soundness.* Next we discuss these two aspects of our work and lay out our motivation for departing from the approach taken by most prior work.

Proving as fast as computing. In the field of delegation of computation (the sequential notion analogous to distributed certification), a great amount of effort has been devoted to constructing provers that add minimal overhead on top of the algorithm whose correctness they aim to certify [44, 3, 26, 24, 10, 49, 50, 11, 35]. This is referred to as "proving as fast as computing". Efficient provers are needed for any practical deployment of a delegation scheme, and therefore designing proof systems where the prover is efficient is a key element in applications such as proofs on the blockchain; for instance, [8, 7] have made great progress in the efficiency of the prover and are used in practice. See [47] for a survey of the subject.

In the distributed setting the need for efficient provers is much the same: in order for distributed certification to serve as a practical mechanism for fault tolerance, we must be able to compute the certificates efficiently. Thus, the goal of our work is *proving as fast as distributed computing*:

> *Given a distributed algorithm $\mathcal{D}$ that runs in polynomial communication rounds and local computation steps, construct a prover that runs alongside $\mathcal{D}$, adding at most a polylogarithmic overhead to the rounds and local computation steps.*

In other words, our goal is to obtain distributed certification schemes where both the verifier and the prover are local (in terms of the overhead they add to the system), in contrast to traditional distributed certification, where only the verifier is a local distributed algorithm, and the prover is all-powerful.

Computational soundness. Most of the work on distributed certification is set in the *information theoretic* world, where the prover and the network nodes are computationally unbounded. The two requirements from a certification scheme for a network property $\mathcal{L}$ are:

- Completeness: if the property $\mathcal{L}$ holds, then there exists a certificate assignment that convinces all nodes to accept; and
- Soundness: if the property $\mathcal{L}$ does not hold, then no certificate assignment convinces all nodes to accept.

Unfortunately, the information-theoretic setting inherits some powerful lower bounds from nondeterministic two-party communication complexity: for example, it is known that some network properties require $\Omega(n^2)$-bit certificates [25], and some simple and natural properties such as proving that the network has a given diameter require $\Omega(n)$-bit certificates, even when the verifier is randomized [22].

This motivates us to consider the following relaxation of the soundness requirement, known as *computational soundness* ([39]):

- Computational soundness: if the property $\mathcal{L}$ does not hold, then no *poly-size prover*[1] can construct a certificate assignment that convinces all nodes to accept.

A proof system that has computational soundness is called an *argument*, and in the distributed setting we call it a *distributed argument*.

One might ask whether computational soundness indeed captures the type of faults from which the network wishes to protect itself. We argue that the answer is *yes*, in most if not all practical scenarios, if one is willing to assume standard cryptographic assumptions hold. The key here is that any fault that could be simulated by an efficient algorithm, cannot break computational soundness, since if it could, that would mean that an efficient algorithm can solve believed-to-be hard problems, such as the discrete logarithm. For example, if we wish to protect against hardware or software faults, then we should demand soundness against all certificates generated generated for an illegal state due to a buggy execution of a distributed algorithm in the network, or against those generated by a buggy version of the honest prover algorithm. But even a buggy prover is still an *efficient algorithm*. Similarly, faults caused by topology changes can also be simulated by an efficient algorithm, which again means that such faults cannot break a computationally sound certification scheme.

We remark that although in this work we weaken the soundness requirement, and construct a local distributed proving algorithm, we still require soundness against *global provers*: the argument that we construct is sound against any polynomial-size "cheating prover" that sees the entire network and tries to produce certificates that fool the network into accepting even though the network is not in a legal state.

Distributed SNARGs. In delegation of computation (the sequential notion analogous to distributed certification), the gold standard is to construct a *succinct non-interactive argument* (SNARG) whose security relies on standard cryptographic hardness assumptions, such as learning with errors [16] or, bilinear maps [48], and decisional Diffie-Hellman [14].[2] A SNARG is a computationally sound proof system in which a polynomial-size prover $\mathcal{P}$ certifies a statement of the form "$x \in \mathcal{L}$," where x is an input of size n and $\mathcal{L}$ is a language, by providing a computationally weak verifier $\mathcal{V}$ with a proof π, of length $|\pi| = \text{polylog}(n)$. The verifier then examines the input x and the proof π, and decides in *linear time* in n whether to accept or reject.[3] It is guaranteed that the honest prover $\mathcal{P}$ can convince the verifier $\mathcal{V}$ to accept any true statement with probability 1 (*perfect completeness*), and at the same time, no poly-size cheating prover can convince the verifier to accept with non-negligible probability (*computational soundness*). The requirement that the proof π be of polylogarithmic length is called *succinctness*.

In recent years, the fruitful line of work on delegation of computation has culminated in the construction of SNARGs for all properties in P [16, 48, 27, 14, 28]. In [2], this was extended to distributed network algorithms. A *distributed* SNARG [2] for a property $\mathcal{L}$ is a computationally sound proof system $(\mathcal{P}, \mathcal{V})$, consisting of

[1] Computational soundness, like other computational hardness notions, models the adversary as a non-uniform machine of polynomial size, as it is at least as strong as randomized.

[2] Throughout this work, we refer to SNARGs for *deterministic computations*, which prove that some polynomial-time computation was executed correctly, and not SNARGs for NP, which are a much stronger cryptographic primitive that is not known to exist under standard cryptographic assumptions.

[3] Technically, the prover and the verifier take as input a security parameter λ, and their running time is polynomial in λ. We defer the discussion of the security parameter to Section 2.

- A prover $\mathcal{P}$, which may or may not be a distributed algorithm (both options were considered in [2]). Given a network graph $G = (V, E)$ and an input assignment $x : V \to \mathcal{X}$ specifying the input $x(v)$ to each node $v \in V$, the prover constructs a *proof* in the form of a certificate assignment $\pi : V \to \{0, 1\}^*$, with each node v receiving a certificate $\pi(v)$ of length $\operatorname{polylog}(n)$ (where $n = |V|$).
- A verification procedure $\mathcal{V}$, which is a one-round distributed algorithm where every node $v \in V$ initially knows its UID, its input $x(v)$, its neighbors in G, and its certificate $\pi(v)$. Each node sends a (possibly different) message on each one of its edges, receives the messages sent by its neighbors, carries out some local computation, and then outputs *accept* or *reject*. The proof is considered accepted if and only if all nodes accept.

It was recently shown in [2] that any network property in P admits a distributed SNARG. Moreover, [2] constructed a *distributed prover*, which allows a polynomial-time distributed algorithm to certify the correctness of its output using certificates of size $\operatorname{polylog}(n)$. However, the prover constructed in [2] is *global*: although it is a distributed algorithm, it collects information from all the nodes of the network, which requires $\Omega(D)$ rounds in networks of diameter D (using messages of polylogarithmic size). This means that in some cases the prover's overhead may eclipse the running time of the distributed algorithm whose correctness it certifies, e.g., if the original algorithm is a local algorithm.

1.1 Our Contribution

In this work, we construct a *fully local distributed argument* that certifies the correctness of any polynomial distributed algorithm. That is, for a polynomial distributed algorithm $\mathcal{D}$, it certifies the following property:

$$\mathcal{L}_{\mathcal{D}} = \left\{ (G, x, y) : \begin{array}{l} \mathcal{D} \text{ produces output } y : V \to \mathcal{Y} \text{ when executed in} \\ \text{the network } G = (V, E) \text{ with input assignment } x : V \to \mathcal{X} \end{array} \right\}.$$

Our construction uses two cryptographic primitives: *collision-resistant hash functions* and *batch arguments for* NP. These are known to exist under several standard cryptographic assumptions: subexponential hardness of Diffie-Hellman [18, 14]; polynomial hardness of learning with errors [1, 15]; and polynomial hardness of bilinear maps [48].

▶ **Theorem 1.** *Assume collision-resistant hash functions and batch arguments for* NP *exist. Then for any distributed algorithm $\mathcal{D}$ that runs in* $\operatorname{poly}(n)$ *rounds local computation time, there is a distributed argument $(\mathcal{P}, \mathcal{V})$ certifying the property $\mathcal{L}_{\mathcal{D}}$, where the prover $\mathcal{P}$ is a distributed algorithm that adds an overhead of* $\operatorname{polylog}(n)$ *rounds to the execution of $\mathcal{D}$, sends* $\operatorname{polylog}(n)$*-bit messages, and produces certificates of length* $\operatorname{polylog}(n)$*, and the verifier $\mathcal{V}$ runs in one round and sends* $\operatorname{polylog}(n)$*-bit messages.*

Our construction relies on *low-diameter network decompositions*, and represents a novel connection between this highly useful primitive and distributed certification.

Applications of our construction. Fully local distributed arguments have several applications. First, they enable efficient certification of local algorithms, where previous constructions either had an overhead of $\Theta(D)$ rounds or produced very long certificates (or both). That is, a distributed algorithm that runs in a small number of rounds but still has high communication complexity (i.e., it uses long messages), could now be certified in a few more rounds, using low communication complexity, and be verified in one round, with one message on each edge. Second, a local prover can be used to efficiently *mend* a proof of correctness, instead

Table 1 Generic distributed certification schemes, and the costs they incur when certifying an algorithm that runs for R rounds and sends B-bit messages in networks with n nodes, maximum degree Δ and diameter D.

	Soundness	Certificate	Verifier Message	Prover Overhead
PLS [34]	Perfect	$R \cdot B \cdot \Delta$	$R \cdot B \cdot \Delta$	No overhead
LCP [25][4]	Perfect	$\Theta(n^2)$	$\Theta(n^2)$	Not distributed
RPLS [23]	Statistical	$R \cdot B \cdot \Delta^2$	$O(\log n)$	1 round
LVD-SNARGs [2][5]	Computational	$\mathrm{poly}(\lambda, \log n)$	$\mathrm{poly}(\lambda, \log n)$	Not distributed
LVD-SNARGs [2]	Computational	$\mathrm{poly}(\lambda, \log n)$	$\mathrm{poly}(\lambda, \log n)$	$O(D)$
This Work	Computational	$\mathrm{poly}(\lambda, \log n)$	$\mathrm{poly}(\lambda, \log n)$	$\mathrm{polylog}\, n$

of re-computing it from scratch when a change occurs in the network. Many distributed algorithms support *local correction* (also called *fixing*, *mending* or *healing*) of their output, that is, if a change in the network causes the output of the algorithm to become incorrect, there is a local procedure that executes only in the area of the network where the change occurred and "fixes" the output of the algorithm (see, e.g., [5, 19, 36, 6, 31] and the references therein). Following the execution of the local correction procedure, our fully local prover can also mend the correctness certificate, by executing the prover to re-certify correctness in the area of the network that was modified by the correction procedure. This application of our work creates a new tie between local correction and distributed certification, areas that both arose originally from fault tolerance and self-stabilization but have drifted apart over time.

Finally, fully local distributed arguments are an important step towards *incrementally verifiable distributed computation*. In sequential computing, incrementally verifiable computation (IVC, [46, 43]) allows for the incremental construction of a certificate of correctness, which is updated after each step taken by the sequential algorithm, and does not require storing the entire trace of the computation in memory. Incrementally verifiable computation is especially relevant in distributed systems, which are often long-lived and reactive. As a first step towards incrementally verifiable distributed computation, it is necessary to have a low-overhead prover that can be called many times during the computation without blocking for a long time, and our construction takes the first step in this direction.

1.2 Related Work

There are several known approaches to obtain generic schemes for certifying the correctness of any given distributed algorithm, although as we mentioned above, most of the focus in prior work has been on the efficiency of the verifier, not the prover. In Table 1 we summarize the tradeoffs that each approach achieves between the length of the certificates, the communication of the verifier, and the complexity of the prover. The table covers only schemes where the verifier runs for one round; it is sometimes possible to trade off certificate size against verifier rounds (see, e.g. [21, 42]), but the total communication over all verification rounds in the information-theoretic setting remains, in general, high. Next, we give a brief overview of each approach.

[4] In [25], the certificate and message size also depend on the size of the input to each node. That is, $\Theta(n^2)$ refers to the case where there is no input to the nodes or the input is of constant size.

[5] In [2], the property is assumed to be in P, and R and B are assumed to be at most polynomial in n.

In the first work to introduce proof labeling schemes [34] it is pointed out that any distributed algorithm can be certified by storing the entire transcript of the algorithm at each node. This can result in long certificates. In [23] it is shown that randomization can be used to exponentially decrease the communication of the verifier, but this comes at the cost of even longer certificates (as well as weakening soundness from *perfect soundness* to *statistical soundness*, where the verifier has some small probability of accepting an invalid proof). Another generic approach is to store a description of the entire network as the certificate at each node [25]. In addition to long certificates, this approach requires the prover to *know* the entire network, which rules out an efficient distributed implementation. However, it has the advantage of not being dependent on the communication complexity of the distributed algorithm to be certified, which could be useful for the certification of highly expensive algorithms.

The first work to introduce computationally sound distributed certification is [2], which showed that any network property in P can be certified using certificates of polylogarithmic length in this setting, assuming the prover knows the entire network. In addition, [2] constructs a generic scheme with a *distributed* prover that can certify the correctness of any given distributed algorithm that runs in polynomial rounds and local computation time. However, the prover in this case requires $O(D)$ rounds in networks of diameter D.

2 Preliminaries

In this section, we describe our network model (which is fairly standard) and the common reference string model, and then go over the two cryptographic primitives used in our construction; hash families with local openings and batch arguments for NP. The description of a batch argument is brief and the full syntax and definition can be found in Appendix A.2. Distributed Merkle trees, which are another existing construct we use, are discussed in Section 3.1, and defined formally in Appendix A.1. Moreover, for lack of space, we defer the full definition of our *fully local distributed* SNARG (fl-DSNARG) to Appendix A.4.

Network model. A synchronous distributed network is modeled as an undirected, connected graph $G = (V, E)$, where the nodes V are the processors participating in the computation, and the edges E represent bidirectional communication links between them. Each network node has a unique identifier v from some UID domain $[\widetilde{n}]$, and we assume that the size of the UID domain is polynomial in the network size n, so that a UID can be encoded in $O(\log n)$ bits. We often conflate the UID of a node with the vertex representing it in the network graph. In each communication round, each node sends a message to each of its neighbors; the nodes then receive the messages sent to them, carry out some internal computation, and then the next round begins. The input to the computation is represented by an assignment $x : V \to \mathcal{X}$, and the output by an assignment $y : V \to \mathcal{Y}$, where $\mathcal{X}, \mathcal{Y}$ are some input and output domains (respectively). Initially, each network node $v \in V$ knows its UID, its input $x(v)$, its neighborhood $N(v)$ in G, and the size n of the network (or a polynomial upper bound on the size, such as $\widetilde{n}$). Each node v eventually produces the output $y(v)$. We restrict attention to algorithms where the round complexity, the message length and the internal computation are polynomial in the size of the network.

The common reference string model and computational hardness. Our work is set in the *common reference string* (CRS) model, which is also the model in which the SNARG constructions of [16, 48, 27, 14] are set. In this model, all parties – in our case, the prover and all the network nodes – have access to a string that is sampled randomly by a trusted setup process, denoted by Gen, which takes a *security parameter* λ in unary representation. (This can be viewed as public randomness.) The security parameter governs the computational resources that must be invested to break the security or soundness of the protocol: we say that a task that involves the CRS is *hard* (or *computationally hard*) if given a CRS sampled using $\mathsf{Gen}(1^\lambda)$, no poly-size (in λ) adversary can succeed in the task, except with *negligible probability* – that is, probability smaller than $1/\lambda^c$ for any constant c. Batch arguments, described below, are defined in the CRS model and their soundness properties hold with respect to such a security parameter.

Collision resistance and hash families with local openings. A *hash family with local openings*, also sometimes known as a *hash tree*, allows a party that holds a vector $(x_1, \ldots, x_n)$ to compute a short hash of the vector, and later to locally *open* specific locations $i \in [n]$, producing a certificate that convinces another party that the value hashed in location i is x_i. The interface is as follows:

- $\mathsf{Gen}(1^\lambda) \to \mathsf{hk}$: a trusted, randomized setup procedure that takes a security parameter λ and outputs a hash key hk. The hash key can be thought of as a descriptor for a hash function chosen at random from a collision-resistant hash family,[6] which will be used in the computation of the hash value and its local openings.
- $\mathsf{Hash}(\mathsf{hk}, x) \to \mathsf{val}$: takes a hash key hk and a bit vector $x \in \{0,1\}^*$, and returns a hash value val.
- $\mathsf{Open}(\mathsf{hk}, x, i) \to (b, \rho)$: takes a hash key hk, a bit vector $x \in \{0,1\}^*$ and an index $i \in [|x|]$, and produces a bit b and an *opening* ρ, which is meant to serve as a certificate that $x_i = b$.
- $\mathsf{Verify}(\mathsf{hk}, \mathsf{val}, i, b, \rho) \in \{0,1\}$: takes a hash key hk, a hash value val, an index i, a bit b, and an opening ρ, and outputs an acceptance bit b. This procedure is meant to be executed by the other party, which does not know the value hashed, and wishes to verify that it has b in location i.

Our requirements of a hash family with local openings are as follows:

- Efficiency and succinctness: the procedures above run in time polynomial in their input, and output values of length at most $\mathrm{poly}(\lambda, \log |x|)$.
- Completeness: for every hk generated by Gen, every input x and every index $i \in [|x|]$, if $\mathsf{val} = \mathsf{Hash}(\mathsf{hk}, x)$ and $(b, \rho) = \mathsf{Open}(\mathsf{hk}, x, i)$, then $\mathsf{Verify}(\mathsf{hk}, \mathsf{val}, i, b, \rho) = 1$.
- Collision-resistance with respect to openings: it is computationally hard, given a hash key hk generated by Gen, to find a hash value val, an index i, and two openings ρ_0, ρ_1, such that both $\mathsf{Verify}(\mathsf{hk}, \mathsf{val}, i, 0, \rho_0) = 1$ and $\mathsf{Verify}(\mathsf{hk}, \mathsf{val}, i, 1, \rho_1) = 1$.

We often describe the existence of an opening ρ such that $\mathsf{Verify}(\mathsf{hk}, \mathsf{val}, i, b, \rho_0) = 1$ by saying that *the hash value* val *opens to* b *in location/index* i.

Merkle tree [38] is a tree-based hash family with local openings that can be constructed from any collision-resistant hash family. Since collision-resistant hash families are known to exist under the assumption of either the hardness of the discrete logarithm problem [18] or the learning with errors problem [1], Merkle trees – and hash families with local openings in general – are also guaranteed to exist under the same assumptions. A Merkle tree over values $(x_1, \ldots, x_n)$ is a binary tree, where the leaves are $x_1, \ldots, x_n$, and each inner node is

[6] A *collision-resistant hash family* is a family of functions $\mathcal{H}$, such that it is computationally hard, given a random function from the family $h \in \mathcal{H}$, to find colliding inputs: x, y such that $h(x) = h(y)$.

the hash of the concatenation of its two children.[7] Merkle trees form the foundation for the distributed Merkle tree construction of [2], which is utilized in our construction of an fl-DSNARG (see Section 3.1).

Batch arguments for NP (BARGs) and their use in SNARG constructions. In the SNARG constructions of [16, 48, 27, 14], to prove that $x \in \mathcal{L}$ for a language $\mathcal{L}$ that is decided by a Turing machine M, the prover essentially proves the following statement:"there exist configurations $\mathsf{cf}_0, \dots, \mathsf{cf}_T$ such that cf_0 is the initial configuration of M on input x, cf_T is an accepting configuration, and for each $i = 0, \dots, T-1$, the machine M transitions from cf_i to cf_{i+1}". This highly-structured statement is a special case of T instances of an *index language*: an NP-language of the form $\mathcal{L} = \{(C, i) : \exists w.\, C(i, w) = 1\}$, where C is a circuit (in this case, verifying the transitions of the Turing machine), and i is an index. To prove such statements, [16, 48, 27, 14] use *batch arguments for* NP (BARGs), which we describe next, as they also serve as the basis for our construction in the current paper.

A batch argument for an index language $\mathcal{L}$ allows a prover to convince a verifier of a conjunction of the form $\varphi(C) = \bigwedge_{i=1}^{k} \exists w_i.\, C(i, w_i) = 1$, where the circuit C is known to both the prover and the verifier, but only the prover knows the witnesses $w_1, \dots, w_k$. To prove this statement, the prover produces a short proof π, which the verifier is able to check. Crucially, the length of the proof π is linear in the length of a single witness $|w_i|$, but only polylogarithmic in the number of statements k.[8]

The BARGs we use in this work, like the BARGs used to construct SNARGs for P, are of a special type, called a *somewhere-extractable* BARG (seBARG). We give here a brief description of a seBARG. See Appendix A.2 for the full syntax and definition. A seBARG allows for the extraction of *one witness* from a convincing proof π, as follows:

- The procedure Gen that generates the CRS for the BARG can be called either in regular mode or in *trapdoor mode*. In trapdoor mode, Gen takes in addition to the security parameter λ an index $i \in [k]$, called the *binding index*. It outputs a pair $(\mathsf{crs}, \mathsf{td})$, where td is a *trapdoor* that can later be used to recover the i-th witness.

  In trapdoor mode, the Gen procedure has a property called *index hiding*: it is computationally hard to *find* the binding index i, given crs. This means that the prover, which is given only crs and not the trapdoor td, "cannot tell" which index we are interested in. In fact, it is hard to even tell whether Gen was called in regular mode or in trapdoor mode, as the distributions of the resulting string crs are computationally indistinguishable.
- The seBARG has an auxiliary *extraction* procedure, $\mathcal{E}(\mathsf{td}, C, \pi)$, which takes a trapdoor td, a circuit C and a proof π, and extracts one witness w.
- The seBARG has the *somewhere argument of knowledge* property: suppose we call Gen in trapdoor mode with a binding index i, and obtain $(\mathsf{crs}, \mathsf{td})$. Given only crs, it is computationally hard to find a proof π that is accepted by the verifier, such that when we extract a witness w_i using $\mathcal{E}(\mathsf{td}, C, \pi)$, we have $C(i, w) \neq 1$. In other words, it is hard for a poly-size adversary to fool the verifier into accepting a proof π if when we extract the i-th witness we find an inconsistency: the witness is not an NP-witness matching index i.

[7] More accurately, the leafs of a Merkle tree over $(x_1, \dots, x_n)$ are hash values of $x_1, \dots, x_n$, taken by a hash function collision-resistant hash family.

[8] Batch arguments for general NP languages allow proving a conjunction of NP statements. A batch argument for an index language allows for a highly efficient verification, as the verifier does not have to read k instances.

3 Technical Overview

In this section, we give a high-level overview of our construction of a fully local distributed SNARG (fl-DSNARG). This overview is somewhat informal, and some technical details are glossed over or omitted. The full construction and analysis are deferred to the full version of this paper.

Given a distributed algorithm $\mathcal{D}$, an input assignment x and an output assignment y, we wish to construct an argument that certifies the execution of each node, to prove that each node v indeed outputs $y(v)$ when $\mathcal{D}$ is executed with input x. We must take into consideration both the local computation of the node and the messages it sends and receives. A naïve approach would be to have each individual node construct a SNARG proof attesting to the internal computation steps that it takes while executing $\mathcal{D}$, but this is not enough: the challenge is that from the perspective of each node, the messages it receives from other nodes are essentially *inputs* to its computation, and the messages it sends are *outputs*. We must verify the consistency of these messages across each edge: messages that node u "attests to sending" to v should indeed be received at node v (i.e., they should be reflected correctly in the proof attesting to v's internal computation).

Unfortunately, while the real input x and output y of the distributed algorithm we are trying to certify are available at verification time, the messages sent by the algorithm are not: we cannot afford to store all messages sent and received as part of the certificate, as this would require far too much space. The solution is to carefully construct a hash of the messages, and use it to have nodes verify that the messages are consistent with the rest of their internal computation.

Recall from Section 2 that current centralized SNARG constructions consist of a batch argument for NP (a BARG) asserting the conjunction of T statements $S_1, \ldots, S_T$, each describing a single transition of a Turing machine. The configurations of the Turing machine are not available explicitly at verification – they are not part of the SNARG proof; instead, only a hash of the configurations is included in the proof. The proof consists (informally) of a batch argument proving that for each step i, the configuration hash opens in the appropriate locations to two configurations $\mathsf{cf}_i, \mathsf{cf}_{i+1}$, such that the Turing machine indeed transitions from cf_i to cf_{i+1}.[9] (The openings are part of the witness encoded inside the batch argument.)

We use a similar idea to handle the messages of the distributed algorithm: we construct multiple *local distributed Merkle trees*, which together are analogous to a hash tree of the messages, in such a way that each node v can compute openings to all the messages it sent or received. Intuitively, the message-hash has a "slot" (an index) for each directed edge $u \to v$ and round r, which is meant to record the message $m_r^{u \to v}$ that is sent from node u to node v in round r. We use the message-hash and the openings to construct two batch arguments: one attesting to the correctness of the internal computation steps at node i, and the second attesting to the consistency between the messages recorded "inside" the internal computation of node v, and the message-hash.

Consistency is verified at both endpoints of every directed edge $u \to v$: node u verifies that the message that it *sent* to node v in round r is indeed recorded in the message-hash in the slot for message $m_r^{u \to v}$, and node v verifies that the message that it *received* from node v in round r is recorded in the message-hash in the slot for message $m_r^{u \to v}$. This ties together the messages sent and received, and ensures that our proof captures the true execution of the distributed algorithm.

[9] Technically, we work with a hash of *a hash* of the configurations (two levels of hashing), so this description is not quite accurate. We give a more detailed one in Section 3.2.

Next we describe distributed Merkle trees, as introduced in [2], and our way of constructing multiply such trees where each one is *local*, using a new notion of *low-diameter edge cover.*

3.1 Local Distributed Merkle Trees

Distributed Merkle trees. As mentioned above, the idea of hashing together all the messages and using this hash to construct succinct arguments for distributed algorithms was introduced in [2], and implemented in the form of a *distributed Merkle tree* (DMT). We introduce DMTs here in a concise manner, see Appendix A.1 for the full syntax and definition.

A DMT is a hash with local openings for a collection of values $\{x_{v\to u}\}_{\{v,u\}\in E}$, one value for every directed edge $v \to u$ such that $\{v,u\} \in E$. The values are initially unordered, but an order will be imposed on them when the DMT is constructed. Initially, each node v knows all values $x_{u\to w}$ such that $u = v$ or $w = v$, that is, all values corresponding to edges that touch v. The DMT is essentially a hash of hashes:

- First, each node v computes a hash $\mathsf{rt}(v)$ (specifically, a Merkle tree) of its own "outgoing" values, $\{x_{v\to w}\}_{w\in N(v)}$; we call $\mathsf{rt}(v)$ the *local root of node v.*
- Then, the nodes compute together a global hash, rt, of the individual hashes $\{\mathsf{rt}(v) : v \in V\}$. We call rt the *global root* of the DMT.

Recall that the values $\{x_{v\to u}\}_{\{u,v\}\in E}$ are initially unordered. As the network constructs the DMT, it imposes an order over the nodes, and each node learns the index $I(v)$ where its own local root is hashed inside the DMT. Since node v constructed its own local root $\mathsf{rt}(v)$, it already knows the index $I_{v\to w}$ where it hashed each value $x_{v\to w}$. We think of the concatenation of these indices, $I(v) \parallel I_{v\to w}$, as *the index of $x_{v\to w}$* in the DMT.

A DMT acts much like a regular Merkle tree over the values $\{x_{v\to u}\}_{\{v,u\}\in E}$. With the information that node v obtains during the construction of the DMT, it can produce an opening from the global root rt to any value $x_{v\to u}$ or $x_{u\to v}$ where $u \in \mathbb{N}(v)$. The DMT serves in [2] as a hash of all the messages sent in the network: each value $x_{v\to u}$ is itself a hash of all the messages that node v sent to node u during the execution of the algorithm.

In [2] it is shown that a DMT can be constructed in $O(D)$ rounds in networks of diameter D, using messages of polylogarithmic length. In other words, the DMT construction algorithm of [2] is *global* in nature: it first constructs a spanning tree of the entire network, and then computes the DMT by aggregating hash values up the tree and propagating openings down the tree.[10] This seems unavoidable, as the DMT is a hash of a collection of values that are initially spread across the entire network. However, one of our main technical contributions is to show that succinct distributed SNARGs do not require a global DMT; rather, we can get away with using a collection of *local* DMT*s*, each applied to a low-diameter subgraph of the original network graph, and thereby reduce the overhead of the prover from $O(D)$ rounds to polylog(n).

Using local DMTs. The key observation that enables us to construct a local prover is that both during the proving stage and at verification, each node requires access only to *its own messages* (sent or received).[11] Thus, there is no need to have a single DMT covering the entire network graph and providing all nodes with a single hash of all the messages; instead,

[10] However, the algorithm in [2] is still more communication-efficient than simply gathering the entire network's transcript in one location to compute the hash tree, as it uses polylogarithmic messages.

[11] The nodes do not actually require access to the messages themselves, but need to be able to verify consistency of them against some hash value.

we can compute many "small" DMTs, each covering a small neighborhood and providing the nodes of that neighborhood with one hash that they can use to access the messages they sent or received within that neighborhood. Moreover, we do not even need all edges of a given node to be covered by the *same* DMT: the crucial property we require is that *every edge must be covered by at least one* DMT, so that the messages that flow across the edge can be incorporated into the certificates of the two nodes at the endpoints of the edge.

With this observation in mind, our goal is to cover all edges of the network by a collection of subgraphs $H_1, \dots, H_k$, with each subgraph H_i maintaining its own DMT. The trade-off that governs our construction is a familiar one for distributed graph algorithms:

- On the one hand, we would like each subgraph H_i to have a *small diameter*, so that we can compute the DMT for the subgraph in a small number of rounds.
- On the other hand, each node should belong to only a small number of subgraphs, as each subgraph corresponds to a separate DMT and increases both the length of the certificate that the node eventually computes and the number (or alternatively, the size) of messages that the node must route during the proving stage, when the DMTs are constructed.

We call the cover $H_1, \dots, H_k$ a *low-diameter edge cover* (defined formally in Appendix A.3), and show below that it can easily be constructed from a *low-diameter decomposition* of G^2, the power-2 graph induced by our network graph G. (In G^2, two nodes $u, v \in V$ are neighbors if and only if their distance in G is at most 2.) We discuss how existing low-diameter decomposition constructions [45, 13] can be extended to handle G^2 while remaining in the CONGEST model in Appendix B.

In each cluster H_i, we compute a DMT over all messages sent over edges of H_i. Each such "local" DMT has a similar structure to the global DMT from [2]. The local DMT for cluster H_i requires $O(\text{diam}(H_i))$ rounds to construct, and this is why we require a small diameter for each cluster.

Constructing a low-diameter edge cover. Suppose we are given an (ℓ, m)*-low diameter decomposition* of $G^2 = (V, E')$: a partition of the nodes V into clusters $U_1, \dots, U_\ell \subseteq V$, and a coloring $c : \{1, \dots, \ell\} \to \{1, \dots, m\}$ of the clusters, such that:

1. The subgraph $G^2[U_i]$ induced by each U_i has diameter at most d, and
2. The coloring c is a proper coloring of the cluster graph: for any $i \neq j$ such that for nodes $u \in U_i$ and $v \in U_j$ there is an edge $\{u, v\} \in E'$ we have $c(i) \neq c(j)$.

Then we can obtain a low-diameter edge cover by defining subgraphs $H_1 = G[S_1], \dots, H_\ell = G[S_\ell]$ that each includes one cluster and all the nodes that are adjacent to it in G:

$$S_i = U_i \cup \{v \in V \ : \ \exists u \in U_i . \{v, u\} \in E\} .$$

For each node v, denote by $C(v) \subseteq \{S_1, \dots, S_\ell\}$ the set of clusters to which v belongs. We have the following properties:

- The diameter of each subgraph H_i is at most $2d + 2$: the original cluster $G^2[U_i]$ has diameter at most d with respect to G^2, which translates to diameter at most $2d$ with respect to G. Adding nodes adjacent to U_i in G increases the diameter to at most $2d + 2$.
- Every edge $\{u, v\} \in E$ is covered by some cluster $H_i = G[S_i]$: since $U_1, \dots, U_\ell$ is a partition of V, there is some $i \in [\ell]$ such that $u \in U_i \subseteq S_i$, and consequently $v \in S_i$. Thus, $\{u, v\}$ is covered by S_i.
- Each node belongs to at most m clusters of the edge cover: if v belongs two clusters S_i and S_j where $i \neq j$, then there exist nodes $u_i \in S_i, u_j \in S_j$ that are both at distance at most 1 from v in G. But this means that u_i and u_j are neighbors in G^2, and hence clusters U_i, U_j are adjacent in G^2, and must have a different color ($c(i) \neq c(j)$). This implies that $|C(v)| \leq m$.

3.2 Constructing the Distributed Argument

To construct our fully local distributed SNARG, we first need to fix a concrete model for the internal computation carried out by the network nodes, as the argument will need to refer to these computation steps. We begin by presenting such a model, and then outline the construction of the $\mathsf{fl\text{-}DSNARG}$.

Modeling polynomial-time distributed algorithms. Consider a distributed algorithm $\mathcal{D}$ that runs in R rounds, with each node taking T local computation steps in each round (including steps required to read or produce messages). For the sake of concreteness, we model $\mathcal{D}$ as a Turing machine[12] $M_{\mathcal{D}}$, which has three tapes:

- The first tape of $M_{\mathcal{D}}$ at node v contains the information available to node v throughout the computation: its UID, its neighbors, and its input $x(v)$.
- On the second tape, $M_{\mathcal{D}}$ writes and receives messages. At the beginning of each round r, the messages that were sent to node v in round $r-1$ appear on this tape; during the round, $M_{\mathcal{D}}$ erases these messages and instead writes the messages that node v sends in round r. For simplicity, we assume in this overview that each message consists of a single bit. (In the full version of this paper, we allow messages to be of polynomial size.)
- The third tape is a work tape, and stores the current internal state of node v.

We denote by $\mathsf{cf}_{r,t}(v)$ the configuration of $M_{\mathcal{D}}$ at node v in the t-th computation step of round r. For each $t < T$, the configuration $\mathsf{cf}_{r,t+1}(v)$ is obtained from $\mathsf{cf}_{r,t}(v)$ by a computation step of $M_{\mathcal{D}}$, representing an internal computation step of node v. However, configuration $\mathsf{cf}_{r+1,1}(v)$ is obtained from $\mathsf{cf}_{r,T}(v)$ by writing on the first tape the messages that v's neighbors sent to node v in round r, as recorded in the third tape of their final round-r configurations, $\{\mathsf{cf}_{r,T}(u) : u \in N(v)\}$. This represents the receipt of these messages by node v at the end of round r.

We refer to the sequence $\mathsf{cf}_{0,0}(v), \ldots, \mathsf{cf}_{R,T}(v)$ as the *trace* of the computation at node v, and denote it by $\mathsf{Trace}(v)$.

Constructing the distributed argument. Fix a distributed algorithm $\mathcal{D}$ where each node executes a Turing machine $M_{\mathcal{D}}$, a network graph $G = (V, E)$, an input assignment $x : V \to \mathcal{X}$ and an output assignment $y : V \to \mathcal{Y}$. Let R, T be the number of rounds and the local computation time of $\mathcal{D}$, respectively. As it runs alongside the original algorithm $\mathcal{D}$, the prover records the execution of $\mathcal{D}$ at each node v: it stores the trace $\mathsf{Trace}(v) = \mathsf{cf}_{0,0}(v), \ldots, \mathsf{cf}_{R,T}(v)$ of the Turing machine $M_{\mathcal{D}}$ executed at node v, and the messages $\{m_r^{v\to u}, m_r^{u\to v}\}_{u\in N(v), r\in[R]}$ sent and received by v (respectively) on each edge $\{v, u\} \in E$ in each round r.[13]

After $\mathcal{D}$ terminates, the prover begins constructing the certificates. The first step is to compute a low-diameter edge cover of the network graph G, as described in Section 3.1. Let $S_1, \ldots, S_\ell \subseteq V$ be the resulting clusters, and for each node v, let $S(v) \subseteq \{1, \ldots, \ell\}$ be the indices of clusters to which node v belongs. In each cluster S_i, we compute a DMT of all messages sent over edges belonging to $G[S_i]$, as described above. In the sequel, we use the notation $(\cdot)_i(v)$ for the DMT associated with cluster i at node v; for example, $\mathsf{rt}_i(v)$ is the local root of node v in the DMT for cluster i.

[12] For simplicity, we assume that all nodes execute the same Turing machine, which takes the UID of the node as input. However, this is not essential; we could have each node v execute a different machine M_v.

[13] We believe that the space requirement of our prover can be reduced to have polylogarithmic overhead on top of the original algorithm $\mathcal{D}$, but this is technically non-trivial, and we defer it to future work.

The remainder of the prover's computation is local: each node uses the information it stored while $\mathcal{D}$ was running, and the DMTs that we constructed, to compute a certificate $\pi(v)$, consists of the following (see Figure 1 for an illustration).

- A hash with local openings $\mathsf{hTrace}(v)$ of the vector $(\mathsf{hCf}_{0,0}(v), \ldots, \mathsf{hCf}_{R,T}(v))$, where each $\mathsf{hCf}_{r,t}(v)$ is itself a hash with local openings of the configuration $\mathsf{cf}_{r,t}(v)$.
- The set $S(v)$ of clusters to which node v belongs.
- For each cluster $i \in S(v)$, the root rt_i of the DMT for cluster S_i, as well as the index and the opening from the root rt_i down to the local root $\mathsf{rt}_i(v)$, which hashes all messages sent by node v over edges belonging to cluster i.
- A BARG proof $\beta^{\mathsf{int}}(v)$ asserting that the *internal computation* of node v is correct, namely, that each configuration $\mathsf{cf}_{r,t+1}(v)$ in the trace of v is obtained from the preceding configuration $\mathsf{cf}_{r,t}(v)$ by a transition of $M_{\mathcal{D}}$.[14] This is a conjunction of $R \cdot (T-1)$ statements, with the (r,i)-th statement asserting (roughly) that there exist two hashed configurations $\mathsf{hCf}, \mathsf{hCf}'$ such that:
  - $\mathsf{hTrace}(v)$ opens to hCf in the index corresponding to step (r,t) of the computation, and to hCf' in the index corresponding to step $(r,t+1)$.
  - The configuration hashes hCf and hCf' are of successive configurations $\mathsf{cf}, \mathsf{cf}'$ (respectively), such that cf' is obtained from cf by one step of $M_{\mathcal{D}}$. This statement is delicate to prove, since it concerns the configurations "under the hash" and not the hashes $\mathsf{hCf}, \mathsf{hCf}'$ themselves (at least not directly), but it can be done using a technique from [29]. In short, it involves proving that the hashes $\mathsf{hCf}, \mathsf{hCf}'$ are of configurations that are only different in one location, and this could be done for a locally-openable hash.
- A BARG proof $\beta^{\mathsf{cons}}(v)$ asserting the consistency of the messages written in v's trace with the messages recorded in the DMTs to which v belongs. This is a conjunction of $R \cdot \widetilde{n}^2$ statements, where $\widetilde{n}$ is the size of the UID space: statement $(r,u,w) \in [R] \times [\widetilde{n}] \times [\widetilde{n}]$ asserts that *if* the edge (u,w) exists in the network, then for each of its ends $v \in \{u,w\}$, *the same* message is recorded in the appropriate index (corresponding to round r and edge (u,w)) of the DMT and trace of node v (which again is u or w).

  In more detail, we require that if the edge (u,w) exists and $v \in \{u,w\}$ is one of its ends, then there exist a message $m \in \{0,1\}$ and a configuration hash hCf such that:
  - The DMT for the cluster covering edge $\{u,w\}$ opens to m in the location corresponding to round r and directed edge (u,w),
  - hTrace opens to the configuration hash hCf in location (r,T) if v is the sender (i.e., $u=v$), or in location $(r+1,1)$ if v is the receiver (i.e., $w=v$), and
  - hCf opens to m in the location where the message sent/received on edge (u,v) is recorded.

  If the edge (u,w) does not exist, or is not adjacent to node v, then the statement (r,u,w) is simply *true* (i.e., it imposes no requirements). The mechanism for checking inside the BARG whether or not the edge (u,w) exists and touches node v is somewhat subtle, and we defer the details to the full version of this paper.

[14] Recall that the transition from step (r,T) to step $(r+1,1)$ involves receiving messages; it is not a local computation step. It must still be attested to, for example to ensure that the internal state of the machine does not change between these two steps, but we omit the details here.

We note that despite the fact that our construction uses multiple local DMTs, the argument presented above is simpler than the argument constructed using the global DMT in [2]: separating the requirements into *internal correctness* and *message consistency*, and creating a separate BARG for each, simplifies both the structure of the argument and the proof of its soundness.

For technical reasons related to the proof of soundness, we actually need two copies of each BARG: $\beta^{\mathsf{int}}(v)^j$ and $\beta^{\mathsf{cons}}(v)^j$, for $j \in \{1,2\}$. Each of the four BARGs uses its own crs, and we will see that this helps us "catch a cheating prover in a lie". This is discussed in Section 3.3 below.

Verifying the certificates. At verification time, each node v informs its neighbors of the clusters $S(v)$ to which it belongs, and also sends a collection $\{(\rho_c(v), I_c(v)) : c \in S(v)\}$ consisting of v's local root and index inside the local DMT for each cluster to which v belongs. This allows each neighbor $u \in N(v)$ to compute the location in the DMT of each message sent on the edge (v,u).

Next, each node v verifies the four BARGs, $\beta^{\mathsf{int}}(v)^j$ and $\beta^{\mathsf{cons}}(v)^j$ for $j = 1, 2$, stored in its certificate $\pi(v)$. At this point it has all the information needed to do so. If the BARG verification succeeds, node v outputs *accept*, and otherwise it outputs *reject*.

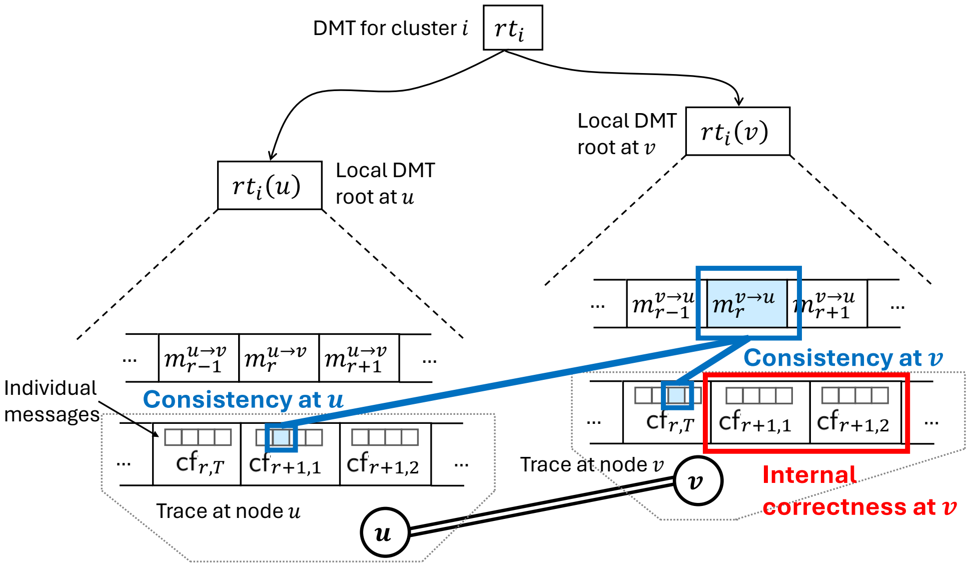

DMT for cluster i
rt_i
Local DMT root at v
$rt_i(v)$
$rt_i(u)$
Local DMT root at u
… $m_{r-1}^{v\to u}$ $m_r^{v\to u}$ $m_{r+1}^{v\to u}$ …
… $m_{r-1}^{u\to v}$ $m_r^{u\to v}$ $m_{r+1}^{u\to v}$ …
Consistency at v
Individual messages
Consistency at u
… $\mathsf{cf}_{r,T}$ $\mathsf{cf}_{r+1,1}$ $\mathsf{cf}_{r+1,2}$ …
… $\mathsf{cf}_{r,T}$ $\mathsf{cf}_{r+1,1}$ $\mathsf{cf}_{r+1,2}$ …
Trace at node v
v
Internal correctness at v
Trace at node u
u

Figure 1 The figure shows the DMT for the cluster i that covers edge $\{u,v\}$, and "under the hash", the messages sent from node u to node v and vice-versa, under the respective local roots $\mathsf{rt}_i(u), \mathsf{rt}_i(v)$. The figure also shows the trace at each node, "under the hash". Inside each configuration, small boxes indicate messages written on the second tape. In configuration $\mathsf{cf}_{r,T}(v)$, these are the messages sent by node v in round r; in configuration $\mathsf{cf}_{r+1,1}(v)$, these are the messages received by node v in round r (and similarly for node u). The internal correctness BARG at node v (in red) asserts that each configuration $\mathsf{cf}_{r,t+1}(v)$ is the successor to $\mathsf{cf}_{r,t}(v)$ according to $M_{\mathcal{D}}$. The consistency BARGs at node v and at node u (in blue) together assert that each message hashed inside the DMT matches the corresponding messages in the traces of u and of v.

3.3 The Soundness of Our Construction

In this section, we give the main ideas for our proof of computational soundness.

Fix a distributed algorithm $\mathcal{D}$, and let $\mathcal{L}_{\mathcal{D}}$ be the language of all annotated graphs (G, x, y) such that when $\mathcal{D}$ executes in the network G with input assignment x, the output it produces is y. Let $(\mathsf{Gen}, \mathcal{P}, \mathcal{V})$ be our fl-DSNARG for the language $\mathcal{L}_{\mathcal{D}}$, as described above.

Recall that computational soundness requires that no poly-size adversary can fool the verifier into accepting the proof of an incorrect statement, except with negligible probability (in the security parameter and in the size of the graph). We capture this requirement in the form of the following experiment, which we call ExpSound, where a poly-size adversary $\mathcal{A}$ tries to break the soundness of the argument:

- A crs is sampled by calling the trusted setup procedure Gen of the fl-DSNARG. In our construction, several of the primitives that we use require a common reference string: the DMT uses a CRS to select a hash function, and the BARGs use their own internal hash functions as well. The Gen procedure of our fl-DSNARG instantiates these common reference strings by calling the Gen procedures of the respective primitives, and returns one value, crs, consisting of all of them together.
- The adversary $\mathcal{A}$ is given crs, and outputs an annotated graph (G, x, y), and a certificate assignment π to the nodes of G.

We say that $\mathcal{A}$ *wins* the experiment if it can produce a network G, an input x and output y such that the algorithm $\mathcal{D}$ does *not* output y on (G, x), and a certificate assignment π that convinces all nodes to accept, nonetheless. If there is a *poly-size* adversary that can win the experiment with non-negligible probability, then soundness is broken.

To prove the soundness of our argument we assume towards contradiction that *there is* a poly-size adversary $\mathcal{A}$ that can win experiment ExpSound. We use $\mathcal{A}$ to construct a poly-size adversary $\mathcal{A}'$ that breaks the soundness of one of our building blocks: the *collision-resistance with respect to openings* property of the hash family, the *index-hiding* property of the BARG, or the *somewhere argument of knowledge* property of the BARG. Since we assume that these properties hold for the primitives we use, this is a contradiction.

We consider each computation step $(r, t) \in [R] \times [T]$ of the distributed algorithm $\mathcal{D}$, and define an experiment $\mathsf{ExpSound}_{r,t}$, which is the same as ExpSound, except that the crs for the two BARGs $\beta^{\mathsf{int}}(v)^1$ and $\beta^{\mathsf{cons}}(v)^1$ is generated in trapdoor mode, binding the crs to index (r, t), while the other two copies, $\beta^{\mathsf{int}}(v)^2$ and $\beta^{\mathsf{cons}}(v)^2$, are set up in regular mode (without a trapdoor). By the *index-hiding* property of the BARG, no poly-size adversary can tell whether the Gen procedure is called in regular mode or in trapdoor mode; therefore, our cheating adversary $\mathcal{A}$ wins the new experiment $\mathsf{ExpSound}_{r,t}$ with almost the same probability that it wins the original experiment, ExpSound, where all four BARGs were set up in regular mode. (If the probability was noticeably different, then we could break the index-hiding property by running $\mathcal{A}$ and checking whether it wins. The noticeable difference between the winning probability for ExpSound and for $\mathsf{ExpSound}_{r,t}$ translates to a noticeable advantage in guessing whether crs was generated in trapdoor mode or not.)

Next we use the *somewhere argument of knowledge* property of the BARG to claim that whenever $\mathcal{A}$ wins the experiment $\mathsf{ExpSound}_{r,t}$, we can use the trapdoor associated with the binding index (r, t) to extract NP-witnesses $w^{\mathsf{int}}_{r,t}(v), w^{\mathsf{cons}}_{r,t}(v)$ to the (r, t)-th statement of the BARGs $\beta^{\mathsf{int}}(v)^1$ and $\beta^{\mathsf{cons}}(v)^1$, again with a very close probability to the original winning probability of $\mathcal{A}$. These witnesses are accepted by the circuit of the respective BARGs.

We would now like to argue that these witnesses reflect the *true* state of the distributed algorithm after the t-th computation step of round r: that is, they match the witnesses that would be generated by an honest prover $\mathcal{P}$, and contain, e.g., the true hash values of internal configurations and messages that the algorithm $\mathcal{D}$ generates at this point in its computation. We will then use the *collision resistance to openings* property of the hash family to reach a contradiction. If we could claim this for *every* $r \in [R]$ and $t \in [T]$, then in particular it would be true for the final state of the network, in step (R, T), where the output y is produced. Since the output is encoded in the internal configuration of the network nodes, whenever the adversary $\mathcal{A}$ wins $\mathsf{ExpSound}_{R,T}$, we can use it to find a collision in the hash of the internal configurations: if $\mathcal{A}$ wins, then for some node $v \in V$, the output $y(v)$ produced by $\mathcal{A}$ does not match the true output $y'(v)$ of the algorithm $\mathcal{D}$. The witness $w^{\mathsf{int}}_{R,T}(v)$ contains a hash $\mathsf{hCf}_{R,T}(v)$ of the false final configuration $\mathsf{cf}_{R,T}(v)'$, which includes the false output $y(v)$. But we know that this witness matches what the honest prover would produce, that is, the hash of the true final configuration $\mathsf{cf}_{R,T}(v)$, including the true output $y'(v)$. Thus, the true configuration $\mathsf{cf}_{R,T}(v)$ and the false configuration $\mathsf{cf}_{R,T}(v)'$ hash to the same value, $\mathsf{hCf}_{R,T}(v)$, and we found a collision.

To prove that the witnesses extracted from the certificates in each experiment $\mathsf{ExpSound}_{r,t}$ are the true witnesses that would be generated by the honest prover, we define *hybrid experiments* $\left\{\mathsf{ExpSound}'_{r,t}\right\}_{(r,t)\in[R]\times[T]}$, where we use two trapdoors: the first two copies of the BARGs are set up with a binding index of (r, t), while the second two copies are set up with a binding index of $(r, t + 1)$. The winning condition for experiment $\mathsf{ExpSound}'_{r,t}$ requires the adversary to output certificates $\pi(v)$ at each node v such that

- All certificates are accepted.
- For each node, upon extracting the witnesses for indices (r, t) and $(r, t + 1)$ from the respective BARGs, all four witnesses are accepted by the respective BARG circuits.[15]
- For each node, the witnesses for index (r, t) are the true witnesses that would be generated by the honest prover. And finally,
- There exists a node where the witnesses for index $(r, t + 1)$ are *not* the true witnesses that would be generated by the honest prover.

Winning this experiment with non-negligible probability again breaks the index-hiding property of the BARG, because it essentially means that the adversary can tell whether the binding index is (r, t), in which case it produces true witnesses matching the honest prover at all nodes, or $(r, t + 1)$, in which case it produces a false witness at some node. Proving this step also relies on the fact that the witnesses are accepted by the BARG circuit, which asserts that the transition from step (r, t) to step $(r, t + 1)$ is legal. This means that if the witness for step (r, t) is the true witness, then either the witness for step $(r, t + 1)$ is also the true witness, or we have broken the *somewhere proof of knowledge* property of the BARG (it accepts, despite the extraction of an inappropriate witness).

After proving that the adversary cannot win experiment $\mathsf{ExpSound}'_{r,t}$ except with negligible probability, we chain together the entire sequence $\mathsf{ExpSound}'_{1,1}, \ldots, \mathsf{ExpSound}'_{R,T}$ and argue that since the adversary does not win *any* of these experiments with non-negligible probability, either it produces false witnesses *for the initial state of the network*, or it produces true witnesses for all computation steps (in which case we are done, as we explained above). However, the prover cannot lie about the initial state of the network without breaking collision resistance, for reasons similar to those we outlined for the final configuration.

[15] Recall that the BARG circuit is simply the circuit that verifies (i, w_i), not to be confused with the BARG verifier, which verifies the BARG proof.

References

1 Miklós Ajtai. Generating hard instances of lattice problems. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 99–108, 1996.

2 Eden Aldema Tshuva, Elette Boyle, Ran Cohen, Tal Moran, and Rotem Oshman. Locally verifiable distributed snargs. In *Theory of Cryptography Conference*, pages 65–90. Springer, 2023.

3 Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the 2017 acm sigsac conference on computer and communications security*, pages 2087–2104, 2017. doi:10.1145/3133956.3134104.

4 B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 268–277, 1991.

5 B. Awerbuch and M. Sipser. Dynamic networks are as fast as static networks. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pages 206–219, 1988.

6 Alkida Balliu, Juho Hirvonen, Darya Melnyk, Dennis Olivetti, Joel Rybicki, and Jukka Suomela. Local mending. In Merav Parter, editor, *Structural Information and Communication Complexity*, pages 1–20, 2022. doi:10.1007/978-3-031-09993-9_1.

7 Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In *45th international colloquium on automata, languages, and programming (icalp 2018)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

8 Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.

9 Aviv Bick, Gillat Kol, and Rotem Oshman. Distributed zero-knowledge proofs over networks. In *SODA*, pages 2426–2458. SIAM, 2022. doi:10.1137/1.9781611977073.97.

10 Jonathan Bootle, Andrea Cerulli, Essam Ghadafi, Jens Groth, Mohammad Hajiabadi, and Sune K Jakobsen. Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 336–365. Springer, 2017. doi:10.1007/978-3-319-70700-6_12.

11 Jonathan Bootle, Alessandro Chiesa, and Jens Groth. Linear-time arguments with sublinear verification from tensor codes. In *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part II 18*, pages 19–46. Springer, 2020. doi:10.1007/978-3-030-64378-2_2.

12 Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof-labeling schemes. *Theoretical Computer Science*, 811:112–124, 2020. doi:10.1016/J.TCS.2018.08.020.

13 Yi-Jun Chang and Mohsen Ghaffari. Strong-diameter network decomposition. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 273–281, 2021. doi:10.1145/3465084.3467933.

14 Arka Rai Choudhuri, Sanjam Garg, Abhishek Jain, Zhengzhong Jin, and Jiaheng Zhang. Correlation intractability and SNARGs from sub-exponential DDH. In *Proceedings of the 43rd Annual International Cryptology Conference, CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 635–668. Springer, 2023. doi:10.1007/978-3-031-38551-3_20.

15 Arka Rai Choudhuri, Abhishek Jain, and Zhengzhong Jin. Non-interactive batch arguments for NP from standard assumptions. In *Proceedings of the 41st Annual International Cryptology Conference, CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 394–423. Springer, 2021. doi:10.1007/978-3-030-84259-8_14.

16 Arka Rai Choudhuri, Abhishek Jain, and Zhengzhong Jin. SNARGs for P from LWE. In *62nd IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 68–79, 2021.

17 Pierluigi Crescenzi, Pierre Fraigniaud, and Ami Paz. Trade-offs in distributed interactive proofs. In *DISC*, volume 146 of *LIPIcs*, pages 13:1–13:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.DISC.2019.13.

18 Ivan Bjerre Damgård. Collision free hash functions and public key signature schemes. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 203–216. Springer, 1987.

19 Shlomi Dolev. *Self-Stabilization.* MIT Press, 2000.

20 Yuval Emek, Yuval Gil, and Shay Kutten. Locally Restricted Proof Labeling Schemes. In *36th International Symposium on Distributed Computing (DISC 2022)*, volume 246, pages 20:1–20:22, 2022. doi:10.4230/LIPICS.DISC.2022.20.

21 Laurent Feuilloley, Pierre Fraigniaud, Juho Hirvonen, Ami Paz, and Mor Perry. Redundancy in distributed proofs. *Distributed Comput.*, 34(2):113–132, 2021. doi:10.1007/S00446-020-00386-Z.

22 Pierre Fraigniaud, Pedro Montealegre, Rotem Oshman, Ivan Rapaport, and Ioan Todinca. On distributed Merlin-Arthur decision protocols. In *SIROCCO*, volume 11639 of *LNCS*, pages 230–245. Springer, 2019. doi:10.1007/978-3-030-24922-9_16.

23 Pierre Fraigniaud, Boaz Patt-Shamir, and Mor Perry. Randomized proof-labeling schemes. *Distributed Computing*, 32:217–234, 2019. doi:10.1007/S00446-018-0340-8.

24 Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S Wahby. Brakedown: Linear-time and field-agnostic snarks for r1cs. In *Annual International Cryptology Conference*, pages 193–226. Springer, 2023. doi:10.1007/978-3-031-38545-2_7.

25 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory Comput.*, 12(1):1–33, 2016. doi:10.4086/TOC.2016.V012A019.

26 Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography with constant computational overhead. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 433–442, 2008. doi:10.1145/1374376.1374438.

27 Yael Kalai, Alex Lombardi, Vinod Vaikuntanathan, and Daniel Wichs. Boosting batch arguments and RAM delegation. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1545–1552, 2023. doi:10.1145/3564246.3585200.

28 Yael Tauman Kalai, Alex Lombardi, and Vinod Vaikuntanathan. Snargs and ppad hardness from the decisional diffie-hellman assumption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 470–498. Springer, 2023. doi:10.1007/978-3-031-30617-4_16.

29 Yael Tauman Kalai, Omer Paneth, and Lisa Yang. How to delegate computations publicly. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 1115–1124. ACM, 2019. doi:10.1145/3313276.3316411.

30 Gillat Kol, Rotem Oshman, and Raghuvansh R. Saxena. Interactive distributed proofs. In *Symposium on Principles of Distributed Computing (PODC)*, pages 255–264, 2018. URL: https://dl.acm.org/citation.cfm?id=3212771.

31 Michael König and Roger Wattenhofer. On local fixing. In *Principles of Distributed Systems*, pages 191–205. Springer International Publishing, 2013. doi:10.1007/978-3-319-03850-6_14.

32 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 26–34, 2006. doi:10.1145/1146381.1146389.

33 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 26–34, 2006. doi:10.1145/1146381.1146389.

34 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 9–18, 2005. doi:10.1145/1073814.1073817.

35 Jonathan Lee, Srinath Setty, Justin Thaler, and Riad Wahby. Linear-time and post-quantum zero-knowledge snarks for r1cs. *Cryptology ePrint Archive*, 2021.

36 Christoph Lenzen, Jukka Suomela, and Roger Wattenhofer. Local algorithms: Self-stabilization on speed. In *Stabilization, Safety, and Security of Distributed Systems*, pages 17–34, 2009. doi:10.1007/978-3-642-05118-0_2.

37 Nathan Linial and Michael Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993. doi:10.1007/BF01303516.

38 Ralph C. Merkle. A certified digital signature. In *Proceedings of the 9th Annual International Cryptology Conference, CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer, 1989. doi:10.1007/0-387-34805-0_21.

39 Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. doi:10.1137/S0097539795284959.

40 Pedro Montealegre, Diego Ramírez-Romero, and Ivan Rapaport. Shared vs private randomness in distributed interactive proofs. *arXiv preprint arXiv:2006.16191*, 2020. arXiv:2006.16191.

41 Moni Naor, Merav Parter, and Eylon Yogev. The power of distributed verifiers in interactive proofs. In Shuchi Chawla, editor, *Symposium on Discrete Algorithms (SODA)*, pages 1096–115, 2020. doi:10.1137/1.9781611975994.67.

42 Rafail Ostrovsky, Mor Perry, and Will Rosenbaum. Space-time tradeoffs for distributed verification. In *International Colloquium on Structural Information and Communication Complexity*, pages 53–70. Springer, 2017. doi:10.1007/978-3-319-72050-0_4.

43 Omer Paneth and Rafael Pass. Incrementally verifiable computation via rate-1 batch arguments. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1045–1056. IEEE, 2022. doi:10.1109/FOCS54457.2022.00102.

44 Noga Ron-Zewi and Ron D Rothblum. Proving as fast as computing: succinct arguments with constant prover overhead. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1353–1363, 2022. doi:10.1145/3519935.3519956.

45 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 350–363, 2020.

46 Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Theory of Cryptography Conference*, pages 1–18. Springer, 2008. doi:10.1007/978-3-540-78524-8_1.

47 Michael Walfish and Andrew J Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2):74–84, 2015. doi:10.1145/2641562.

48 Brent Waters and David J. Wu. Batch arguments for NP and more from standard bilinear group assumptions. In *Proceedings of the 42nd Annual International Cryptology Conference, CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 433–463. Springer, 2022. doi:10.1007/978-3-031-15979-4_15.

49 Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*, pages 733–764. Springer, 2019. doi:10.1007/978-3-030-26954-8_24.

50 Jiaheng Zhang, Tianyi Liu, Weijie Wang, Yinuo Zhang, Dawn Song, Xiang Xie, and Yupeng Zhang. Doubly efficient interactive proofs for general arithmetic circuits with linear prover time. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 159–177, 2021. doi:10.1145/3460120.3484767.

A Full Syntax, Formal Definitions and Statements

A.1 Distributed Merkle Trees

We give here the full definition of a distributed Merkle tree, adopted from [2], with a minor change we discuss bellow.

Syntax. An efficient distributed Merkle tree DMT is associated with a recursive hash family with local openings

$$\mathsf{MT} = (\mathsf{MT.Gen}, \mathsf{MT.Hash}, \mathsf{MT.Open}, \mathsf{MT.Verify})$$

and consists of the following algorithms:

$\mathsf{Gen}(1^\lambda) \to \mathsf{hk}$**.** A randomized algorithm that takes as input the security parameter λ and outputs a hash key $\mathsf{hk} = \mathsf{MT.Gen}(1^\lambda)$.

$\mathsf{DistMake}(\mathsf{hk}; G; x) \to \{(\mathsf{val}_v, \mathsf{rt}_v, I_v, \rho_v, \beta_v)\}_{v \in V(G)}$**.** A distributed algorithm that executes in a distributed network G, with all nodes receiving the same hash key hk, and each node $v \in V(G)$ initially holding a collection of inputs $x(v) = \{x_{v\to u}\}_{u \in N(v)}$ (one input $x_{u\to v}$ for each neighbor $u \in N(v)$). The output at each node v consists of:

- A hash value val_v, which is the same at all nodes,
- A local MT-root rt_v,[16]
- An index $I_v \in \{0,1\}^*$,
- An opening path ρ_v, and
- A set β_v of openings $(\rho_{v\to u})$ of index and opening path for every neighbor $u \in N(v)$.

▶ **Definition 2** (DMT). *A* DMT *is required to satisfy the following properties:*

Well-formedness.
- *All nodes $v \in V(G)$ output the same value* val_v*,*
- *All indices I_v are of length $c \cdot \lceil \log n \rceil$, for some constant c,*

MT-***functionality.*** *Fix a hash key* hk*, a network G of size n and input assignment to it $x : V(G) \to \{0,1\}^*$, where for every $v \in V(G)$, $x(v) = \{x_{v\to u}\}_{u \in N(v)}$, such that for every edge $\{v,u\} \in E(G)$, $x_{v\to u} \in \{0,1\}^\ell$. Let*

$$\left\{(\mathsf{val}_v, \mathsf{rt}_v, I_v, \rho_v, p_v, \hat{F}_v, \beta_v)\right\}_{v \in V(G)} = \mathsf{DistMake}(\mathsf{hk}, G, x),$$

where $\beta_v = \{\rho_{v\to u}\}_{u\in N(v)}$. For each directed edge (v,u), let $\mathsf{Index}(v,u) = I_v \parallel \mathsf{id}(u)$*, and* $\mathsf{Opening}(v,u) = \rho_v \parallel \rho_{v\to u}$*. We say that the* DMT *satisfies* MT-functionality *if for every such output, there exists a constant c and a vector $\vec{x}$ of length at most $\leq 2^{c\cdot\lceil \log n \rceil + \lceil \log \widetilde{n} \rceil + \lceil \log \ell \rceil}$ (where $\widetilde{n}$ denotes the size of the UID domain) such that:*
- *For every $v \in V(G)$ and $u \in N(v)$ we have $\vec{x}_{\mathsf{Index}(v,u)} = x_{v\to u}$,*
- *For every $v \in V(G)$,* $\mathsf{val}_v = \mathsf{MT.Hash}(\mathsf{hk}, \vec{x})$*,*
- *For every $v \in V(G)$ and $u \in N(v)$ we have:* $(\vec{x}_{v\to u}, \mathsf{Opening}(v,u)) = \mathsf{MT.Open}(\mathsf{hk}, \vec{x}, \mathsf{Index}(v,u))$*.*

Efficiency. *At each node, the local computation executed by* $\mathsf{DistMake}$ *runs in time* $\mathrm{poly}(\lambda, n, m)$*.*

[16] Throughout this section and the sequel, we use both val and rt to denote MT-values, which are also themselves MT-roots (the construction is recursive). We use val to denote a "final" value, the root of the entire network, which is later exposed to the algorithm using the DMT; we typically use rt for intermediate values handled inside the distributed Merkle.

Low round complexity and low communication complexity. DistMake *runs in* $O(D)$ *synchronized communication rounds on networks of diameter* D*, and uses messages of length* $\mathrm{poly}(\lambda, \log n)$*.*

▶ Remark 3. In [2], the set β_v returned from the algorithm DistMake also contain indices $\{I_v \parallel I_{v\to u}\}_{u\in N(v)}$, and the MT-functionality property is defined with respect to $\mathsf{Index}(v, u) = I_v \parallel I_{v\to u}$, where $I_v \parallel I_{v\to u}$ is the port number of u as represented in the node v. In this work, we simplify this by considering the UIDs instead of port numbers of the nodes. This means that the hash value val now depends on the size of the UID domain $\widetilde{n} = |\mathcal{U}|$, where it used to depend on the maximal degree, but this does not come with a meaningful cost as (1) we assume $\widetilde{n} = \mathrm{poly}(n)$ and (2) the dependency (of previously Δ and now $\widetilde{n}$) is *logarithmic*.

▶ **Theorem 4** ([2]). *For every recursive hash family with local openings, there exists a respective distributed Merkle tree.*

A.2 Somewhere Extractable Batch Arguments (seBARGs)

Syntax. A seBARG for index language consists of the following algorithms:

$\mathsf{Gen}(1^\lambda, k, 1^s, i) \to (\mathsf{crs}, \mathsf{td})$. A randomized setup procedure that takes a security parameter λ, the number of statements k, the size of the circuit 1^s, and an optional index i, and generates a common reference string crs and if provided an index i, a trapdoor td.

$\mathcal{P}(\mathsf{crs}, C, w_1, \ldots, w_k) \to (b, \pi)$. A polynomial-time prover algorithm that takes the crs, a circuit C and a list of witnesses $w_1, \ldots, w_k$, and outputs a bit b and a proof π.

$\mathcal{V}(\mathsf{crs}, C, \pi) \to b$. A polynomial-time verification algorithm that takes the crs, a circuit C, and a proof π and outputs an acceptance bit.

$\mathcal{E}(\mathsf{td}, C, \pi) \to w_i$. A polynomial-time extraction algorithm that takes a trapdoor td, a circuit C, and a proof π, and outputs a witness w_i.

▶ **Definition 5** (seBARG). *A* seBARG *satisfies the following requirements.*

Succinctness. *The length of the* crs *and of the proof* π *is at most* $\mathrm{poly}(s, \lambda, \log k)$*.*

Verifier Efficiency. *The verifier runs in time* $\mathrm{poly}(s, \lambda, \log k)$*.*

Completeness. *For any* $\lambda \in \mathbb{N}$ *and* $s = s(\lambda)$ *of size at most* 2^λ*, for any circuit* $C : [k] \times \{0,1\}^m \to \{0,1\}$ *of size at most* s*, any witnesses* $w_1, \ldots, w_k \in \{0,1\}^m$ *and any index* $i^* \in [k]$

$$\Pr\left[\ \mathcal{V}(\mathsf{crs}, C, \pi) = 1 \ \middle|\ \begin{array}{l} (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, k, 1^s, i^*) \\ \pi \leftarrow \mathcal{P}(\mathsf{crs}, C, w_1, \ldots, w_k) \end{array} \right] = 1.$$

Index hiding. *For any poly-size adversary* $\mathcal{A}$ *and polynomials* $k = k(\lambda)$ *and* $s = s(\lambda)$*, there exists a negligible function* $\mathrm{negl}(\cdot)$ *such that for every* $\lambda \in \mathbb{N}$

$$\Pr\left[\begin{array}{l} i_0, i_1 \in [k] \\ \mathcal{A}(\mathsf{crs}) = b \end{array} \ \middle|\ \begin{array}{l} (i_0, i_1) \leftarrow \mathcal{A}(1^\lambda) \\ b \leftarrow \{0,1\} \\ (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, k, 1^s, i_b) \end{array} \right] \leq \frac{1}{2} + \mathrm{negl}(\lambda).$$

Somewhere argument of knowledge. *For any poly-size adversary $\mathcal{A}$, polynomials $k = k(\lambda)$ and $s = s(\lambda)$, and index $i^* = i^*(\lambda) \in [k(\lambda)]$, there exists a negligible function* $\text{negl}(\cdot)$ *such that for every $\lambda \in \mathbb{N}$*

$$\Pr\left[\begin{array}{c|l} \mathcal{V}(\mathsf{crs}, C, \pi) = 1 & (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, k, 1^s, i^*) \\ \wedge\ C(i^*, w) = 0 & (C, \pi) \leftarrow \mathcal{A}(\mathsf{crs}) \\ & w \leftarrow \mathcal{E}(\mathsf{td}, C, \pi) \end{array}\right] \leq \text{negl}(\lambda).$$

▶ **Theorem 6** ([15, 48, 27, 14]). seBARG*s for* NP*, and in particular, for the index languages, exist assuming either: (1)* LWE*, (2)* DLIN*, or (3) subexponential* DDH*.*

A.3 Low-Diameter Edge Cover

For a graph $G = (V, E)$ and a mapping S from V to subsets of $\mathcal{U}$, denote by T_S the image of S (that is, the set $\{t \in \mathcal{U} \mid \exists v \in V : \ t \in S(v)\}$), and for every $t \in T_S$, denote by V_t the set of nodes which have t in their image: $V_t^S = \{v \in V \mid t \in S(v)\}$.

▶ **Definition 7** ((D, s)-edge-cover). *For a graph G, we say a mapping $S : V \to \mathcal{U}$ is an edge cover of G if for every edge $\{v, u\} \in E$, we have $S(v) \cap S(u) \neq \emptyset$.*

We say S is diameter-D if for every $t \in T$, we have that the graph induced by V_t^S, $G[V_t]$ is of strong-diameter at most D.

We say S is s-succinct if for every node $v \in V$, we have $|S(v)| \leq s$.

▶ Remark 8. We remark that unlike the classical definition of graph decomposition, here we think of the clusters from the point of view of the nodes; and for that reason define the edge-cover to be a mapping from nodes to all of the sets it belongs to, rather than simply a set of subsets of the graph nodes.

▶ **Theorem 9.** *There exists a* $(\text{polylog}(n), \text{polylog}(n))$*-edge-cover algorithm in the* CONGEST *model.*

A.4 Fully Local Distributed SNARG

We give here the full definition of a *fully local distributed* SNARG (fl-DSNARG), which is mostly adopted from [2], with the only difference being the improved efficiency requirement from the prover.

Syntax. A locally verifiable distributed SNARG with a round-efficient distributed prover for a distributed algorithm $\mathcal{D}$ and corresponding graph language $\mathcal{L}_{\mathcal{D}}$ consists of the following algorithms.

$\mathsf{Gen}(1^\lambda, n) \to \mathsf{crs}$. A randomized algorithm that takes as input a security parameter 1^λ and a graph size n, and outputs a common reference string crs.

$\mathcal{P}(\mathsf{crs}; G; x) \to (y, \pi)$. A *distributed algorithm* that runs in the network G, where all of the nodes have access to the common reference string crs obtained from Gen, and each node $v \in V(G)$ inputs $x(v)$, and outputs (1) an assignment of outputs $y : V(G) \to \{0,1\}^*$ of $\mathcal{D}$ when executed in G, and (2) an assignment of proofs $\pi : V(G) \to \{0,1\}^*$.

$\mathcal{V}(\mathsf{crs}; G; x, \pi) \to b$. A *distributed decision algorithm* that takes as a common input to the entire network a common reference string crs, executes in the network G, where each node $v \in V(G)$ is assigned with an input $x(v)$ and a proof $\pi(v)$, and outputs acceptance bits $b : V \to \{0,1\}^*$.

▶ **Definition 10** (fl-DSNARG). *Let $\mathcal{D}$ be a distributed algorithm, and let $\mathcal{L}_{\mathcal{D}}$ be its corresponding graph language. An* fl-DSNARG $(\mathsf{Gen}, \mathcal{P}, \mathcal{V})$ *for $\mathcal{D}$ must satisfy the following properties:*

Completeness. *For any $(G, x) \in \mathcal{L}_{\mathcal{D}}$,*

$$\Pr\left[\ \mathcal{V}(\mathsf{crs}; G; x, \pi) = 1 \ \middle|\ \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, n) \\ \pi \leftarrow \mathcal{P}(\mathsf{crs}; G; x) \end{array} \right] = 1.$$

Soundness. *For any poly-size algorithm $\mathcal{P}^*$ and polynomial $n = n(\lambda)$, there exists a negligible function* $\operatorname{negl}(\cdot)$ *such that*

$$\Pr\left[\begin{array}{l} (G, x) \notin \mathcal{L}_{\mathcal{D}} \\ \wedge\ \mathcal{V}(\mathsf{crs}; G; x, \pi) = 1 \end{array} \ \middle|\ \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, n) \\ (G, x, \pi) \leftarrow \mathcal{P}^*(\mathsf{crs}) \end{array} \right] \le \operatorname{negl}(\lambda).$$

Succinctness. *The* crs *and the proof $\pi(v)$ at each node v are of length at most* $\operatorname{poly}(\lambda, \log n)$.

Verifier efficiency. *$\mathcal{V}$ runs in a single synchronized communication round, during which each node sends a (possibly different) message of length* $\operatorname{poly}(\lambda, \log n)$ *to each neighbor. At each node v, the local computation executed by $\mathcal{V}$ runs in time* $\operatorname{poly}(\lambda, |\pi(v)|, |x(v)|, \deg(v)) = \operatorname{poly}(\lambda, n)$.

Prover efficiency. *$\mathcal{P}$ adds an overhead of* $\operatorname{polylog}(n)$ *communication rounds to the rounds of $\mathcal{D}$, where in each of these rounds, each node sends a message of length* $\operatorname{poly}(\lambda, \log n)$ *to each neighbor. At each node, the local computation executed by $\mathcal{P}$ runs in time* $\operatorname{poly}(\lambda, n)$.

The following theorem states the existence of fl-DSNARG, assuming the existence of the ingredients we used, to complement Theorem 1.

▶ **Theorem 11.** *Assume the existence of a (D, c)-edge-cover algorithm in the* CONGEST *model, a distributed Merkle tree, and a somewhere extractable argument of knowledge for* NP.

Then, for every distributed algorithm $\mathcal{D}$ that runs in polynomial rounds and local computation time, there exists an fl-DSNARG.

B G^2 Strong-Diameter Decomposition in the CONGEST Model

We require a $(\operatorname{polylog}(n), \operatorname{polylog}(n))$-decomposition algorithm that satisfies the following properties:

- It is a *strong-diameter* decomposition algorithm,
- it is in the CONGEST model, and
- it can be extended to graph powers while remaining in the CONGEST model.

While the first two requirements are rather obvious, the last one may seem trivial given the second requirement, but it is in fact more delicate. It is true that given an algorithm in the LOCAL model, to simulate its execution on G^2, is rather simple; each node could start by collecting its distance-2 neighborhood and then simulate each step of the original algorithm as if it was operating on G^2, while suffering a factor of 2 in the number of rounds. However, this does not generally work in the CONGEST model, as the distance-2 neighborhood of each node might be much larger than the number of connections it can use to collect the information.

In [45], a CONGEST algorithm for *weak-diameter* is constructed using the building block of *weak-diameter ball-carving* algorithm. Their weak-diameter ball-carving is then extended to be simulatable on G^k in the CONGEST model for any constant k, while preserving the round complexity. It then uses a classical CONGEST reduction from *ball-carving* to graph decomposition [37], where the ball-carving algorithm is executed $\log n$ times.

In [13], a strong-diameter decomposition is constructed using a transformation from weak-diameter ball carving to strong-diameter ball carving in the CONGEST model, following by the same classical reduction from ball-carving to decomposition. Their transformation satisfies the property that if the original algorithm runs in $\operatorname{polylog} n$ rounds and produces $\operatorname{polylog} n$-diameter clusters, then the new algorithm also runs in $\operatorname{polylog} n$ rounds and produces $\operatorname{polylog} n$-diameter clusters (with different polynomial dependencies in $\log n$). Then, combining this transformation with the weak-diameter CONGEST ball-carving of [45], they obtain a $\operatorname{polylog} n$ rounds $\operatorname{polylog} n$ strong-diameter ball-carving in the CONGEST model, followed by a corresponding strong-diameter decomposition in the CONGEST model.

Since the weak-diameter ball carving of [45] could be simulated on G^2 in the CONGEST model, to see that the strong-diameter decomposition of [13] could be simulated on G^2 in the CONGEST model it remains to show that their weak-diameter to strong diameter ball-carving transformation could be also simulated in G^2 in the CONGEST model. We observe that the transformation of [13] uses communication between the nodes in the following two ways, which both could be simulated on G^2 in the CONGEST model:

- Counting the number of nodes in a cluster, by gathering information over Steiner trees. This could be simulated for G^2 since each node has to transfer only a *number* of nodes, where this number is still bounded by n in G^2, and so could be described in $O(\log n)$ bits.
- Computing a radius around a node v such that the ratio between the number of nodes in the cluster within that radius around v and the number of nodes beyond that radius exceeds some parameter. This is done by growing a BFS tree around v and gathering the number of nodes within each distance. Here as well, we have that nodes only transfer numbers, which are bounded by n, and thus their description is of size $O(\log n)$.

A Knowledge-Based Analysis of Intersection Protocols

Kaya Alpturer
Princeton University, NJ, USA

Joseph Y. Halpern
Cornell University, Ithaca, NY, USA

Ron van der Meyden
UNSW Sydney, Australia

Abstract

The increasing wireless communication capabilities of vehicles creates opportunities for more efficient intersection management strategies. One promising approach is the replacement of traffic lights with a system wherein vehicles run protocols among themselves to determine right of way. In this paper, we define the *intersection problem* to model this scenario abstractly, without any assumptions on the specific structure of the intersection or a bound on the number of vehicles. Protocols solving the intersection problem must guarantee safety (no collisions) and liveness (every vehicle eventually goes through). In addition, we would like these protocols to satisfy various optimality criteria, some of which turn out to be achievable only in a subset of the contexts. In particular, we show a partial equivalence between eliminating unnecessary waiting, a criterion of interest in the distributed mutual-exclusion literature, and a notion of optimality that we define called *lexicographical optimality*. We then introduce a framework to design protocols for the intersection problem by converting an *intersection policy*, which is based on a global view of the intersection, to a protocol that can be run by the vehicles through the use of knowledge-based programs. Our protocols are shown to guarantee safety and liveness while also being optimal under sufficient conditions on the context. Finally, we investigate protocols in the presence of faulty vehicles that experience communication failures and older vehicles with limited communication capabilities. We show that intersection protocols can be made safe, live and optimal even in the presence of faulty behavior.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Computer systems organization → Dependable and fault-tolerant systems and networks; Computing methodologies → Reasoning about belief and knowledge

Keywords and phrases Intersection management, Autonomous vehicles, Distributed algorithms, Epistemic logic, Fault tolerance

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.2

Related Version *Full Version*: `https://arxiv.org/abs/2408.09499` [2]

Funding *Kaya Alpturer*: Research supported by AFOSR grant FA23862114029. Work done in part while the author was studying at Cornell University.
Joseph Y. Halpern: Research supported in part by AFOSR grant FA23862114029, NSF grant FMitF 2319186, ARO grant W911NF-22-1-0061, and MURI grant W911NF-19-1-0217.
Ron van der Meyden: The Commonwealth of Australia (represented by the Defence Science and Technology Group) supported this research through a Defence Science Partnerships agreement.

1 Introduction

Traffic lights can slow down traffic significantly, due to their lack of responsiveness to real-time traffic. If vehicles can communicate with each other (which is already quite feasible with today's wireless technology), the door is open for improved protocols, where vehicles can determine right of way among themselves, depending on traffic conditions, and thereby

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 2; pp. 2:1–2:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

significantly increase throughput at an intersection. In this paper, we formally define the *intersection problem*: we assume that agents can communicate with each other via radio broadcasts, and design protocols that take advantage of this communication to allow agents to go through the intersection while satisfying *safety* (no collisions) and *liveness* (every vehicle eventually goes through). In addition, we consider *optimal* protocols, which means, roughly speaking, that the protocol allows as many vehicles as possible to go through the intersection at any given time. Finally, we consider the extent to which we can tolerate communication failures and (older) vehicles that are not equipped with wireless, so cannot broadcast messages. (It turns out that these two possibilities can be dealt with essentially the same way.)

While the inefficiencies of traffic-light-based intersection management have long been recognized [7], prior approaches have mainly focused on specific intersection scenarios [15, 16] or relied on executing leader-election protocols without considering communication failures [9, 10]. Furthermore, the protocols have often been evaluated based on simulations of specific intersections, rather than being proved correct [10, 15]. Given the implications of this problem for traffic safety, as well as its potential for greatly improving energy efficiency and productivity, there is a need for formal guarantees on both correctness and optimality.

To the best of our knowledge, prior work did not consider optimality, especially in the presence of various faults. In designing these protocols, to the extent possible, we want them to be robust to a variety of communication failures, such as contexts with *crash* failures,[1] where an agent may fail by ceasing to participate in the protocol at a given time, and *omission* failures, where arbitrary messages can fail to be broadcast.

Epistemic logic has been shown to provide a high-level abstraction that can be used to design distributed protocols independent of particular assumptions on the communication environment and type of failures [8]. Most analyses of distributed-computing problems that use epistemic logic have used full-information protocols to derive time-optimal algorithms, at the cost of large message size and memory requirements. Given the limitations of wireless networks, it is also desirable to bound the amount of information that needs to be exchanged between agents, while still ensuring that the formal guarantees are still met. To address this, following [1], we separate the part of the protocol that determines what information is exchanged between the agents, and the part that determines what action to take based on the agent's information. Thus, when we consider optimality, we do so with respect to protocols that limit information exchange in the same way.

We model the intersection problem as the following scenario. There is a (possibly infinite) set of agents $Ag \subseteq \mathbb{N}$. The intersection has ℓ lanes, represented by $\mathcal{L} = \{0, \dots, \ell - 1\}$. The set of lanes is partitioned into a set of lanes $\mathcal{L}_{in} = \{0, \dots, k - 1\}$, where $1 < k < \ell$ by which vehicles approach the intersection, and a set of lanes $\mathcal{L}_{out}$ by which they depart from the intersection. Each lane in $\mathcal{L}_{in}$ has a queue of agents waiting to go through the intersection; at each point in time at most one agent arrives at each of these queues. A *move* through the intersection is represented by a pair $(l_s, l_t) \in \mathcal{L}_{in} \times \mathcal{L}_{out}$. Intuitively, executing (l_s, l_t) means that the agent arrives through lane l_s and departs through lane l_t. The symmetric relation $\mathcal{O} \subseteq (\mathcal{L}_{in} \times \mathcal{L}_{out})^2$ describes which moves of the agents are compatible; $((l_s, l_t), (l'_s, l'_t)) \in \mathcal{O}$ means that both (l_s, l_t) and (l'_s, l'_t) can be executed in the same round. Broadcasts have a limited range, given by $\rho > 0$. We assume that, provided there are no failures, all broadcasts sent by an agent i will be received by all agents that are within a distance ρ of i. The

[1] We follow the distributed-algorithms literature's interpretation of "crash failure" here: it is not meant to imply a physical collision.

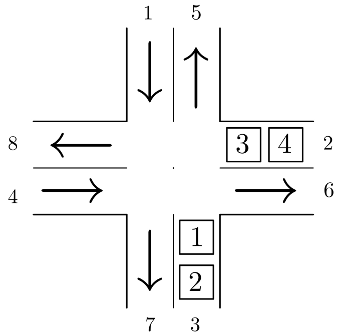


Figure 1 An intersection with $\mathcal{L} = \{1, \ldots, 8\}$ where $\mathcal{L}_{in} = \{1, \ldots, 4\}$ and $\mathcal{L}_{out} = \{5, \ldots, 8\}$. There are currently 4 agents that have arrived in incoming lanes 2 and 3.

problem is then to maximize the rate at which cars move through the intersection while guaranteeing safety (it is never the case that agents with incompatible moves go through the intersection simultaneously) and liveness (all agents that arrive at the intersection eventually move through it). The problem can be thought of as a generalization of distributed mutual exclusion, where the intersection is the critical section.

The rest of the paper is organized as follows: In Section 2, we briefly review the knowledge-based framework of [8]. In Section 3, we modify the information-exchange model of [1] and introduce the sensor model. Section 4 defines models for the adversary which determine the arrival schedule of vehicles and communication failures. Section 5 combines the information-exchange and the adversary model, fully specializing the general model of Section 2 to intersections. Section 6 introduces the various notions of optimality we care about such as eliminating unnecessary waiting and lexicographical optimality. In Section 7, intersection policies are introduced as a global view of the intersection. Section 8 proves a construction that results in an optimal policy even with failures, and explores applications of the construction in two limited-information contexts. Section 9 concludes with a discussion on connections to distributed mutual exclusion. We defer most proofs to the full paper.

2 Reasoning about knowledge

In order to reason about the knowledge of the vehicles in the intersection problem, we use the standard runs-and-systems model [8]. An interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ consists of a system $\mathcal{R}$, which is a set of runs, and an *interpretation* $\pi : \mathcal{R} \times \mathbb{N} \to \mathcal{P}(Prop)$. Each *run* $r : \mathbb{N} \to L_e \times \Pi_{i \in Ag} L_i$ describes a particular infinite execution of the system where $r(m)$ is the global state of the system in run r at time m. The global states consist of an environment state drawn from L_e and local states for each agent i drawn from each L_i. The local state of agent i at point (r, m) is denoted $r_i(m)$. We call a run and time pair (r, m) a *point*. The interpretation π describes which atomic propositions hold at each point in a system $\mathcal{R}$.

We write $\mathcal{I}, (r, m) \models \phi$ if the formula ϕ holds (is satisfied) at point (r, m) in interpreted system $\mathcal{I}$. A formula ϕ is *valid in an interpreted system* $\mathcal{I}$, denoted $\mathcal{I} \models \phi$, if ϕ holds at all points in $\mathcal{I}$; the formula ϕ is *valid* if it is valid in all interpreted systems. Satisfaction of formulas is inductively defined as follows:

- $\mathcal{I}, (r, m) \models p$ iff $p \in \pi(r, m)$.
- $\mathcal{I}, (r, m) \models \phi \wedge \phi'$ iff $\mathcal{I}, (r, m) \models \phi$ and $\mathcal{I}, (r, m) \models \phi'$.
- $\mathcal{I}, (r, m) \models \neg\phi$ iff $\mathcal{I}, (r, m) \not\models \phi$.

- $\mathcal{I}, (r, m) \models K_i\phi$ iff $\mathcal{I}, (r', m') \models \phi$ for all points (r', m') such that $r_i(m) = r'_i(m')$.
- $\mathcal{I}, (r, m) \models \Diamond\phi$ iff for some $m' \geq m$, $\mathcal{I}, (r, m') \models \phi$.
- $\mathcal{I}, (r, m) \models \bigcirc\phi$ iff $\mathcal{I}, (r, m + 1) \models \phi$.

Agent i *knows* a formula ϕ at (r, m) if $\mathcal{I}, (r, m) \models K_i\phi$. Intuitively, agent i knows ϕ if ϕ holds at all points where agent i has the same local state. We say that agent i considers the point (r', m') *possible* at point (r, m) if $r_i(m) = r'_i(m')$. The relation $\sim_i$ is defined as $(r, m) \sim_i (r', m')$ iff $r_i(m) = r'_i(m')$. The formula $\bigcirc\phi$ means that ϕ holds at the next time, and $\Diamond\phi$ means that ϕ holds eventually. In later sections, we formalize how interpreted systems for the intersection problem are specified.

3 Information-exchange protocols

Our framework for modeling limited information exchange is similar to that used by Alpturer et al. [1] to analyze consensus protocols, but we make a number of changes due to the differences in our setting. Here, global states represent not just the result of messages sent between the agents, but also facts about a changing external world, from which the agents obtain sensor readings (e.g., information about their own position and that of nearby vehicles, from GPS, visual, lidar, or radar sensors). We modify the definition of information-exchange protocols from [1] to accommodate these sensor readings. Specifically, assume that we are given a set L_e of environment states. Define a *sensor model* for L_e to be a collection of mappings $\mathcal{S} = \{\mathscr{S}_i\}_{i \in Ag}$, where $\mathscr{S}_i : L_e \to \Sigma_i$ maps states of the environment to a set Σ_i of possible sensor readings for agent i.

An *information-exchange protocol* $\mathcal{E}$ for agents Ag and sensor model $\mathcal{S}$ is given by the collection $\{\mathcal{E}_i\}_{i \in Ag}$ consisting of a local information-exchange protocol $\mathcal{E}_i$ for each agent i. Each local information-exchange protocol $\mathcal{E}_i$ is a tuple $\langle L_i, Mem_i^{init}, A_i, M_i, \mu_i, \delta_i \rangle$, where

- $L_i = Mem_i \times \Sigma_i$ is a set of local states, where each local state consists of a memory state from a set Mem_i and a sensor reading from Σ_i;
- $Mem_i^{init} \subseteq Mem_i$ is a set of initial memory states. (Typically, there might be a single initial memory state, containing information such as the agent's identity.)
- M_i is the set of messages that can be sent by agent i;
- $\mu_i : L_i \times A_i \times \Sigma_i \to M_i \cup \{\perp\}$ is a function mapping a local state s, an action a, and a sensor reading o to the message to be broadcast (intuitively, $\mu_i(s, a, o) = m$ means that when agent i performs action a in state s and obtains new sensor reading o, the information-exchange protocol broadcasts the message m to the other agents; if $m = \perp$, then no message is sent by i);
- $\delta_i : L_i \times A_i \times \mathcal{P}(\cup_{j \in Ag} M_j) \to Mem_i$ is a function that updates the local memory as a function of the previous local state (comprised of the previous memory state and the previous sensor reading), an action, and a set of messages received.

An *action protocol* P for an information-exchange protocol $\mathcal{E}$, is a tuple $\{P_i\}_{i \in Ag}$ containing, for each agent i, $P_i : L_i \to A_i$ mapping the local states L_i for agent i in $\mathcal{E}_i$ to actions in A_i.

4 Adversary model

Intersection protocols need to operate in an environment with several forms of nondeterminism: how messages are broadcast through the environment, failures of transmitters and receivers, and the arrival pattern of vehicles. We model these aspects of the environment in terms of an adversary.

The precise physics of the intersection may affect how broadcasts are transmitted through the environment. Rather than attempt to model Euclidean distances and obstacles, we abstract the effects of these factors on transmission. A *transmission environment* is a relation $T \subseteq (\mathcal{L}_{in} \times \mathbb{N})^2$. Intuitively, $((\ell, p), (\ell', p')) \in T$ represents that, provided the agents' transmitters and receivers do not fail, a message broadcast by an agent at position p in lane ℓ, will be received by an agent at position p' in lane ℓ'. Transmission environments encode our assumption that the communication range is ρ. We make one assumption about this relation: that for all $\ell, \ell' \in \mathcal{L}_{in}$, we have $((\ell, 0), (\ell', 0)) \in T$. That is, messages broadcast by an agent at the front of some lanes are received (barring failure) by all agents that are at the front of any lane.

An *adversary model* $\mathcal{F}$ is a set of adversaries; formally, an *adversary* is a tuple $\alpha = (\tau, T, F_t, F_r)$, where $\tau : Ag \to \mathbb{N} \times \mathcal{L}_{in} \times \mathcal{L}_{out}$, T is a transmission environment, $F_t : \mathbb{N} \times Ag \to \{0, 1\}$, and $F_r : \mathbb{N} \times Ag \to \{0, 1\}$. Intuitively, τ is an *arrival schedule*, which describes when each agent arrives in the system (i.e., enters a queue), its lane of arrival, and its intended departure lane. The function F_t represents failures of agents' transmitters and the function F_r represents failures of agents' receivers. $F_t(k, i) = 1$ means that if i tries to broadcast in round $k + 1$ (i.e., between time k and time $k + 1$), then the broadcast will be sent to all agents within range (i.e., within ρ of i), and perhaps others; similarly, $F_r(k, j) = 1$ means that j receives all broadcasts sent in round $k + 1$ by agents within range (but again, it may receive other broadcasts as well). Thus, a broadcast by agent i in round $k + 1$ is received by a j within range of i in round $k + 1$ iff $F_t(k, i) = F_r(k, j) = 1$. The function τ describes when agents arrive in the system (which we assume is under the control of the adversary). In more detail, if $\tau(j) = (k, (l_1, l_2))$, then at time k, agent j arrives in the system on lane l_1 with the intention of departing on lane l_2. We assume that τ is *conflict-free* in the sense that, for all agents $i \neq j$, if $\tau(i) = (k, (l_1, l_2))$ and $\tau(j) = (k, (l_1', l_2'))$, then $l_1 \neq l_1'$. This ensures that we do not have a conflict of two agents wanting to enter the same queue for lane l_1 simultaneously. (Exactly how this mutual exclusion of queue entry is assured is outside the scope of the model. One way that it may come about is that vehicles approaching the intersection are already ordered along an approaching lane.)

We consider adversary models that involve the following types of failures:

- No failures (*NF*): the set of all adversaries (τ, T, F_t, F_r) where $F_r(k, i) = F_t(k, i) = 1$ for all $i \in Ag$ and $k \in \mathbb{N}$.
- Crash failures (*CR*): the set of all adversaries (τ, T, F_t, F_r) where for all $i \in Ag$ and $k \in \mathbb{N}$, (1) $F_t(k, i) = 0$ implies $F_t(k', i) = 0$ for all $k' > k$, and (2) $F_r(k, i) = 1$ for all k and i.
- Sending omissions (*SO*): the set of all adversaries (τ, T, F_t, F_r) where for all $i \in Ag$ and $k \in \mathbb{N}$, $F_r(k, i) = 1$.

An adversary model $\mathcal{F}$ has a *fixed transmission environment* if all adversaries in $\mathcal{F}$ include the same transmission environment T. We believe that our techniques can be applied without change to the general omissions case.

5 Intersection Contexts

A *context* is a triple $(\mathcal{E}, \mathcal{F}, \pi)$ consisting of an information-exchange protocol $\mathcal{E}$, an adversary model $\mathcal{F}$, and an interpretation π. To deal with intersections, we restrict information-exchange protocols and interpretations so that they satisfy certain conditions. $(\mathcal{E}, \mathcal{F}, \pi)$ is an *intersection context* if it satisfies the following conditions:

- The set of environment states L_e consists of states of the form $s_e = (\alpha, t, q_1, \ldots, q_{|\mathcal{L}_{in}|}, \mathit{done})$ where $\alpha \in \mathcal{F}$ is an adversary, $t \in \mathbb{N}$ is a time, for each approach lane $l \in \mathcal{L}_{in}$, q_l is a queue (list) of agents, intuitively the ones who have lane i and not yet departed, and a set $\mathit{done} \subseteq \mathit{Ag}$, representing the agents that have already passed through the intersection.
- The sensor model, in principle, could be defined to include information from a large variety of sensors and information sources, such as GPS, in-road or road-side beacons, lidar, radar, or vision systems. We start with a minimal location-based sensor model, and leave it open for other fields to be added. Our minimal sensor model $\mathcal{S} = \{\mathscr{S}_i\}_{i \in Ag}$ is defined so that the sensor function $\mathscr{S}_i$ maps environment states to tuples of the form $\langle \mathit{front}_i, \mathit{lane}_i, \mathit{intent}_i \rangle$, where $\mathit{front}_i \in \{0, 1\}$, $\mathit{lane}_i \in \mathcal{L}_{in} \cup \{\bot, \top\}$, and $\mathit{intent}_i \in \mathcal{L}_{out}$. For $s_e = (\alpha, t, q_0, q_1, \ldots, q_{|\mathcal{L}_{in}|}, \mathit{done})$, we have $\mathscr{S}_i(s_e) = \langle \mathit{front}_i, \mathit{lane}_i, \mathit{intent}_i \rangle$, where if τ is the arrival schedule in the adversary α,
  - pos_i maps from global states to $\mathbb{N} \cup \{\bot, \top\}$; $\mathit{pos}_i(s_e) = \top$ if $i \in \mathit{done}$, $\mathit{pos}_i(s_e) = k$ if there exists a queue ℓ such that i is the kth position in queue q_ℓ (with the front of the queue counted as position 0), and $\mathit{pos}_i(s_e) = \bot$ otherwise. (It follows from the state dynamics given below that i is in at most one queue, so pos_i is well-defined.)
  - $\mathit{front}_i = 1$ iff $\mathit{pos}_i(s_e) = 0$,
  - if i is in the queue q_ℓ for lane ℓ, then $\mathit{lane}_i = \ell$; if $i \in \mathit{done}$ then $\mathit{lane}_i = \top$; and if $i \notin \mathit{done}$ then $\mathit{lane}_i = \bot$.
  - if $\tau(i) = (k, (l, l'))$ then $\mathit{intent}_i = l'$.

  We have modelled an agent's intended departure lane intent_i as being received from the environment since, from the point of view of protocol design, this is part of the adversary.
- The set of possible actions of agent i in $\mathcal{E}_i$ is $A_i = \{\mathtt{go}, \mathtt{noop}\}$. Intuitively, `go` represents that action of the agent making its planned move through the intersection. This action can be performed by agent i only if i is at the front of its queue. The action `noop` represents that the agent does not move, unless it is either scheduled for arrival in some queue, or in some position in a queue but not at the front, and the position before it is being vacated, in which case it advances in the queue.
- A global state is a tuple of the form $(s_e, \{s_i\}_{i \in Ag})$, where $s_e \in L_e$ and $s_i \in L_i$ for each agent $i \in \mathit{Ag}$. An *initial* global state has
  - $s_e = (\alpha, t, q_1, \ldots, q_{|\mathcal{L}_{in}|}, \mathit{done})$, where $t = 0$, each queue q_l is empty, and done is the empty set, and
  - for each agent $i \in \mathit{Ag}$, the local state $s_i = (m_i, \mathscr{S}_i(s_e))$ where $m_i \in \mathit{Mem}_i^{init}$ is an initial memory state.
- π interprets the following atomic propositions based on the global state in the obvious way: front_i, $\mathit{lane}_i = l$ for $l \in \mathcal{L}_{in}$, $\mathit{intent}_i = l$ for $l \in \mathcal{L}_{out}$, $\mathit{pos}_i = k$ for $k \in \mathbb{N} \cup \{\bot, \top\}$.

Given an intersection context $\gamma = (\mathcal{E}, \mathcal{F}, \pi)$ and a protocol P, we construct an interpreted system $\mathcal{I}_{\gamma,P} = (\mathcal{R}_{\mathcal{E},\mathcal{F},P}, \pi)$ representing all the possible behaviours of the protocol P in context γ. The set $\mathcal{R}_{\mathcal{E},\mathcal{F},P}$ of runs consists of all runs r that satisfy the following properties:

- The initial state $r(0)$ of r is an initial global state.
- For each $k \in \mathbb{N}$, the global state $r(k+1) = (s'_e, \{s'_i\}_{i \in Ag})$ is determined from $r(k) = (s_e, \{s_i\}_{i \in Ag})$ by a procedure in which the order of events is as follows. First, the agents decide their actions (to go through the intersection or not). They then perform these actions, causing the queues to be updated; any newly arriving agents are also added to the queues in this step. The agents then take a sensor reading, from which they obtain new information about their position. This new information may be included in the message that an agent broadcasts. Finally, each agent updates its memory state, based

on their previous local state, the action performed, and the messages that were broadcast in the current round and received by the agent. We then proceed to the next round. Formally, state transitions are determined by the following procedure:

  - First, each agent i determines its action $P_i(s_i)$ according to the protocol P.
  - If $s_e = (\alpha, m, q_1, \ldots, q_{|\mathcal{L}_{in}|}, \mathit{done})$, then we take $s'_e = (\alpha, m+1, q'_1, \ldots, q'_{|\mathcal{L}_{in}|}, \mathit{done}')$, defined as follows. Note that the adversary α is the same in s'_e, and the time m is incremented. Each queue q'_ℓ is obtained from q_ℓ by the following operations:
    * If $q_\ell(0) = i$ and $P_i(s_i) = \mathtt{go}$, then let q''_ℓ be the result of dequeueing agent i from q_ℓ. Otherwise $q''_\ell = q_\ell$.
    * If $\tau(i) = (m+1, (l_1, l_2))$ for any agent i, then we define $q'_\ell = \mathit{enqueue}(i, q''_\ell)$, otherwise $q'_\ell = q''_\ell$. (Recall that such an i is unique, by assumption on τ.)
  - Finally, we take done' to be the result of adding to the set done all agents i who were at the front of any queue in s_e such that $P_i(s_i) = \mathtt{go}$.
- Next, for each agent i, we obtain a new sensor reading $\mathscr{S}_i(s'_e)$ of the updated state s'_e of the environment. Using this sensor readings, each agent i constructs the message $m_i = \mu_i(s_i, P_i(s_i), \mathscr{S}_i(s'_e))$, which it broadcasts.
- For each agent i, we determine the set of messages B_i^m that the agent receives in round $m+1$. If agent i is not in any queue in state s'_e, or $F_r(m, i) = 0$ (agent i's receiver fails in round $m+1$) then $B_i^m = \emptyset$. Otherwise, for each agent i that is in a lane queue, let ℓ_i be the lane it is in and p_i its position in the queue. We define B_i^m to be the set of messages m_j for which both $((p_j, \ell_j), (p_i, \ell_i)) \in T$ (j's transmission can be heard by agent j, given their positions) and $F_t(m, j) = 1$ (j's transmitter does not fail in this round.)
- Finally, if $s_i = (u_i, \mathscr{S}_i(s_e))$, then $s'_i = (u'_i, \mathscr{S}_i(s'_e))$, where $u'_i = \delta_i(s_i, P_i(s_i), B_i^m)$. (Note that we use the old sensor reading $\mathscr{S}_i(s_e)$ to determine the new memory state, but not the new sensor reading $\mathscr{S}_i(s'_e)$, since the latter will be visible to the agent in its new local state s'_i.)

P is an *intersection protocol* for context $\gamma = (\mathcal{E}, \mathcal{F}, \pi)$ if the following are valid in $\mathcal{I}_{\gamma,P}$ for all $i, j \in \mathit{Ag}$ where $i \neq j$, where going_i is an abbreviation for $\mathit{front}_i \wedge \bigcirc\neg\mathit{front}_i$.

- **Validity:** $\mathit{going}_i \Rightarrow \mathit{front}_i$.
- **Safety:** $(\mathit{going}_i \wedge \mathit{going}_j) \Rightarrow ((\mathit{lane}_i, \mathit{intent}_i), (\mathit{lane}_j, \mathit{intent}_j)) \in \mathcal{O}$.
- **Liveness:** $\mathit{front}_i \Rightarrow \Diamond\mathit{going}_i$.

Intuitively, **Validity** states that an agent does not move through the intersection unless it is at the front of the queue in its lane. **Safety** states that if two agents go through the intersection at the same time, their moves are compatible and do not cause a collision. (Note that the semantics of the action go has been defined so as to ensure that an agent makes its planned move, and not any other.) **Liveness** states that an agent eventually gets to make its move through the intersection. (The model implicitly assumes that vehicles do not have mechanical failures and block other vehicles in their lane.)

6 Unnecessary waiting and optimality

One desirable property of an intersection protocol is that it never makes agents wait unnecessarily. Eliminating unnecessary waiting is also a criterion that has been considered in the distributed mutual-exclusion literature [14]. Intuitively, unnecessary waiting occurs if, given what happens in a certain run r, there is a point where if an agent had gone through the intersection instead of waiting, safety would not be violated. In this section, we define a notion of optimality that captures eliminating unnecessary waiting.

We first give some definitions to define unnecessary waiting and a domination-based notion of optimality. For an intersection context γ and protocol P,

- $GO(r,m)$ is the set of agents that go through the intersection in round $m+1$, that is, the agents i with $\mathcal{I}_{\gamma,P},(r,m) \models \mathit{going}_i$.
- $\mathcal{I}_{\gamma,P},(r,m) \models \mathit{safe\text{-}to\text{-}go}_i$ if $\mathcal{I}_{\gamma,P},(r,m) \models \mathit{pos}_i = 0$ and for all agents $j,k \in GO(r,m) \cup \{i\}$ where $j \neq k$, $(\mathit{lane}_j(r,m), \mathit{intent}_j(r,m))$ and $(\mathit{lane}_k(r,m), \mathit{intent}_k(r,m))$ are compatible moves according to $\mathcal{O}$.
- For a run r of a protocol P in context γ, define $\mathit{gotime}(r,i)$ to be the time $m \in \mathbb{N}$ such that $\mathcal{I}_{\gamma,P},(r,m) \models \mathit{going}_i$, and ∞ if there is no such time.
- $\mathit{front}(r,m)$ is the set of agents that are in front of each queue, that is, the agents i with $\mathit{front}_i(r,m) = 1$.

▶ **Definition 1** (unnecessary waiting). *An intersection protocol P has* unnecessary waiting *with respect to an intersection context γ if there exists $i \in Ag$ and point (r,m) such that $\mathcal{I}_{\gamma,P},(r,m) \models \mathit{safe\text{-}to\text{-}go}_i$ and $i \notin GO(r,m)$.*

▶ **Definition 2** (corresponding runs). *Given action protocols P, P' and context γ, two runs $r \in \mathcal{I}_{\gamma,P}$ and $r' \in \mathcal{I}_{\gamma,P'}$* correspond *if $r(0) = r'(0)$.*

Intuitively, corresponding runs have the same adversary, so agents arrive at the intersection in the same sequence and at the same times in the two runs. We use this notion to define the following notion of one protocol being better than another if it always ensures a faster flow of traffic.

▶ **Definition 3** (domination). *An action protocol P* dominates *action protocol P' with respect to a context γ if for all pairs of corresponding runs $r \in \mathcal{I}_{\gamma,P}$ and $r' \in \mathcal{I}_{\gamma,P'}$, all $i \in Ag$, we have $\mathit{gotime}(r,i) \leq \mathit{gotime}(r',i)$. If P dominates P' but P' does not dominate P, then P* strictly dominates *P'.*

▶ **Definition 4** (optimality). *An intersection protocol P is* optimal *with respect to an intersection context γ if there is no intersection protocol P' that strictly dominates P with respect to γ.*

Our goal is to connect the notions of unnecessary waiting and optimality. The following result shows that the absence of unnecessary waiting is sufficient for optimality.

▶ **Proposition 4.** *If an intersection protocol P has no unnecessary waiting with respect to an intersection context γ then P is optimal with respect to γ.*

From here on, we consider contexts that require some conditions on broadcasting. This is because if not enough information is exchanged or adversaries are too powerful, we cannot have a protocol that avoids unnecessary waiting. To see why, consider a setting where the intersection has two incoming lanes and one outgoing lane, each agent has access to a global clock, and the information-exchange protocol does not send any messages. While a correct protocol exists that uses the global clock to determine when an agent at the front of a queue can proceed to the intersection (essentially, we use the global clock to simulate a traffic light, and have the agents proceed in turns), unnecessary waiting cannot be eliminated, simply because the agents do not exchange enough information to rule out safety violations.

However, even with full information exchange where each agent broadcasts its entire local state in each round and records every broadcast it receives, the converse of Proposition 4 still does not hold. A protocol may have unnecessary waiting and still be optimal even with full information exchange.

▶ **Proposition 4.** *There exists an intersection context γ with full information exchange and no failures and an intersection protocol P such that P has unnecessary waiting and is optimal with respect to γ.*

Proposition 4 suggests that the definition of optimality doesn't exactly capture the lack of unnecessary waiting. We thus consider another definition that we call *lexicographic optimality.*

▶ **Definition 5** (lexicographical domination). *An action protocol P* lexicographically dominates *action protocol P' with respect to a context γ if for all corresponding runs $r \in \mathcal{I}_{\gamma,P}$ and $r' \in \mathcal{I}_{\gamma,P'}$, either $GO(r,m) = GO(r',m)$ for all times m or, at the first time m when $GO(r,m) \neq GO(r',m)$, we have $GO(r',m) \subsetneq GO(r,m)$. If P lexicographically dominates P' but P' does not lexicographically dominate P, then P* strictly *lexicographically dominates P'.*

▶ **Definition 6** (lexicographic optimality). *An intersection protocol P is* lexicographically optimal *with respect to an intersection context γ if there is no intersection protocol P' that strictly lexicographically dominates P with respect to γ.*

▶ **Proposition 6.** *If an intersection protocol P has no unnecessary waiting with respect to an intersection context γ, then P is lexicographically optimal with respect to γ.*

The following result provides a partial converse to Proposition 6.

▶ **Proposition 6.** *If an intersection protocol P is lexicographically optimal with respect to an information context γ with full information exchange and no failures, then P has no unnecessary waiting with respect to γ.*

While considering a full-information context shows that lexicographic optimality captures the condition on unnecessary waiting better, it is also possible to get a similar result in a context with much less information exchange, even without a global clock.

We say that an intersection context $\gamma = (\mathcal{E}, \mathcal{F}, \pi)$ is *sufficiently rich* if $\mathcal{E}$ satisfies the following conditions:

- In round m, if agent i is going to be at the front of some lane at time m, then i broadcasts a message encoding $lane_i$, $intent_i$. (Note that we are here using the fact that in agent's message in round m can incorporate the effect of its round m action. Thus, if an agent i moves to the front of the queue for some lane in round m, then i will sense that it is at the front of the queue, and i can send a message in round m saying that it is about to be at the head of the queue for its lane.)
- Each agent records the $(lane, intent)$ pair for each agent in the front of a queue, and either no agents in the queue other than those at the front broadcast, or agents at the front of a queue tag their messages to indicate that they are at the front of their queue.

Intuitively, if an intersection context is sufficiently rich, in the round m that an agent i reaches the front of the queue for some lane, it knows about all other agents that are in the front of their queues at time m, and knows their intentions (if there are no failures).

▶ **Lemma 6.** *If γ is a sufficiently rich intersection context with no failures, P is an intersection protocol, and $front_i(r,m) = 1$, then*

$$\mathcal{I}_{\gamma,P} \models \forall l \in \mathcal{L}_{in}(K_i(\exists j \in Ag\ \exists l' \in \mathcal{L}\ (front_j \wedge lane_j = l \wedge intent_j = l') \vee \\ K_i(\forall j \in Ag\ (lane_j \neq l)))).$$

Given a sufficiently rich intersection context γ, all protocols that we care about will depend only on what the agents hear from agents at the front of each queue. We say that an intersection protocol P *depends only on agents in the front of their queues* in intersection context $\gamma = (\mathcal{E}, NF, \pi)$ if, for all $i \in Ag$, the following condition holds: for all pairs s_i, s'_i of possible local states of agent i drawn from L_i in $\mathcal{E}$, if $\mathit{front}(r,m) = \mathit{front}(r,m')$, then $P_i(r_i(m)) = P_i(r'_i(m'))$. Note that this condition makes sense only in a sufficiently rich intersection context in the no-failures setting, since otherwise an agent may not know which agents are at the front of their queues, so its protocol cannot depend on this fact.

▶ **Proposition 6.** *Let γ be a sufficiently rich intersection context with no failures. If an intersection protocol P is lexicographically optimal with respect to γ and P depends only on agents in the front of their queues, then P has no unnecessary waiting with respect to γ.*

7 Intersection policies

Intuitively, an *intersection policy* describes which moves are permitted, as a function of a history describing what happened in the run until that point in time (in particular, the nondeterministic choices that have been made by the adversary up to that moment of time), but excluding details of the agent's local states and protocol.

We will use intersection policies as a tool to design standard protocols that solve the intersection problem. Roughly, the methodology is the following. Initially, we will design an intersection policy σ that guarantees safety and liveness for agents complying with σ. We will then find standard intersection protocols that implement a knowledge-based program using σ. Finally, we will show that every intersection protocol can be obtained in this way.

A history captures the nondeterministic choices made by the adversary up to some moment of time. Given an adversary $\alpha = (\tau, T, F_t, F_r)$ for a context γ and natural number $m \in \mathbb{N}$, define the *choices of α in round $m+1$* to be the tuple $\alpha_m = (\tau^m, T, F_r^m, F_t^m)$, where $\tau^m = \{(i,\ell,\ell') \in Ag \times \mathcal{L}_{in} \times \mathcal{L}_{out} \mid \tau(i) = (m+1,\ell,\ell')\}$, and for $a = r$ and $a = t$, the function $F_a^m : Ag \to \{0,1\}$ is defined by $F_a^m(i) = F_a(m,i)$. (Recall that the transmission environment T is fixed for the run, so the same T applies in each round.) An *adversary history* is a finite sequence of such tuples; for an adversary α and time m, define $H(\alpha,m) = \langle \alpha_0, \ldots, \alpha_{m-1} \rangle$. (If $m = 0$, $H(\alpha,m)$ is the empty sequence.) Given a context γ, $\mathcal{H}_\gamma$ is the set of all adversary histories $H(\alpha,m)$ such that α is an adversary for γ and $m \geq 0$. If r is a run of context γ with adversary α, we also write $H(r,m)$ for $H(\alpha,m)$.

▶ **Definition 7** (intersection policy). *An* intersection policy *for a context γ is a mapping $\sigma : \mathcal{H}_\gamma \to \mathcal{P}(\mathcal{L}_{in} \times \mathcal{L}_{out})$.*

Intuitively, an intersection policy says which moves are *permitted* in the given round. An agent at the front of a queue for lane ℓ *may* go if its intent is to make move to lane ℓ' and the move (ℓ,ℓ') is permitted. (However, in contexts with failures, the agent may fail to go because it does not know that its move is permitted.)

An infinite sequence $h_0, h_1, \ldots$ is *feasible* in a context γ if there exists an adversary α of γ such that $h_m = H(\alpha,m)$ for all $m \geq 0$. An intersection policy σ for a context γ is *correct* for a context γ if it satisfies the following specification:

- **Conflict-free:** For all histories $h \in \mathcal{H}_\gamma$, and agents $i \neq j$, if $(l_i,l'_i),(l_j,l'_j) \in \sigma(h)$ then $(l_i,l'_i,l_j,l'_j) \in \mathcal{O}$.
- **Fairness:** For all feasible infinite sequences of histories $h_0,h_1,h_2\ldots$, all moves $(\ell,\ell') \in \mathcal{L}_{in} \times \mathcal{L}_{out}$, and all $m \geq 0$, there exists $m' \geq m$ such that $(\ell,\ell') \in \sigma(h_{m'})$.

Intuitively, an intersection policy σ is conflict-free if σ never permits a conflicting set of moves to occur simultaneously. An intersection policy σ is fair if, in every feasible infinite sequence of histories, σ permits every possible move infinitely often. A context γ is *σ-aware* for an intersection policy σ if, for all protocols P for γ, agents i, lanes $\ell \in \mathcal{L}_{in}$ and $\ell' \in \mathcal{L}_{out}$, we have $\mathcal{I}_{\gamma,P} \models ((\ell,\ell') \in \sigma \wedge \mathit{lane}_i = \ell) \Rightarrow K_i((\ell,\ell') \in \sigma)$.

▶ **Example 8.** A simple correct intersection policy is a cyclic traffic light. Suppose that the set of all moves $\mathcal{L}_{in} \times \mathcal{L}_{out}$ is partitioned into a collection $S_0, \ldots, S_{K-1}$, such that each set S_k is a compatible set of moves. Then the intersection policy defined on histories h by $\sigma(h) = S_{|h| \bmod K}$ is easily seen to be correct (whatever the context γ). Clearly, every synchronous context is σ-aware for this policy.

▶ **Example 9.** A more complicated intersection policy is one that prioritizes certain lanes if they contain specific agents (e.g., an ambulance). Suppose that $\mathcal{A} \subseteq Ag$ is a finite set of higher-priority agents. Consider the intersection policy that allows moves given by a cyclic traffic-light policy unless there is an agent in $\mathcal{A}$ that has arrived and is yet to make a move. In that case, the policy runs the traffic-light policy restricted to lanes containing higher-priority agents. This requires considering past moves permitted by the policy and the adversary history to determine the state of the queues. In a context with no failures, synchrony, and a transmission environment such that the presence of a higher-priority agent is known by agents in the front, we get σ-awareness.

Given an intersection policy σ, consider the following knowledge-based program $\mathbf{P}^\sigma$:

Program $\mathbf{P}^\sigma_i$.

```
if $K_i(front_i \wedge (lane_i, intent_i) \in \sigma)$ then go
else noop
```

Here the formula $(lane_i, intent_i) \in \sigma$ is satisfied at a point (r, m) if we have $(lane_i(r, m), intent_i(r, m)) \in \sigma(H(r, m))$.

An action protocol P *implements* a knowledge-based program of the form "if $K_i\phi$ then go else noop" in a context γ if, for all points (r, m) of $\mathcal{I}_{\gamma,P}$, we have $P_i(r_i(m)) =$ go iff $\mathcal{I}_{\gamma,P}(r, m) \models K_i\phi$. (See [8] for the definition for more general program structures.)

We immediately get the following.

▶ **Proposition 9.** *For every synchronous context γ and intersection policy σ for γ, there exists a behaviorally unique*[2] *P implementing the knowledge-based program $\mathbf{P}^\sigma$ with respect to γ. If σ is a correct intersection policy with respect to γ, then every implementation P of the knowledge-based program $\mathbf{P}^\sigma$ with respect to γ satisfies safety and validity.*

Proposition 9 provides a way of deriving an intersection protocol from an intersection policy. We can also show that every intersection protocol can be derived from some intersection policy in this way.

▶ **Proposition 9.** *If P is a protocol satisfying validity and safety then there exists a conflict-free intersection policy σ for γ such that P implements $\mathbf{P}^\sigma$ with respect to γ.*

▶ **Definition 10** (efficient intersection policies). *An intersection policy σ for a context γ is* efficient *if for all points $h \in \mathcal{H}_\gamma$, we have that $\sigma(h)$ is a maximal conflict-free set of moves.*

8 A Knowledge-Based Program with Lexicographically Optimal Implementations

We would like to have a way to derive lexicographically optimal protocols under a range of failure assumptions. Moreover, we want these protocols to be fair to all agents, even if there are agents present that are not. To satisfy these goals, we start with an intersection policy σ

[2] "Behavioral uniqueness" here means that any two implementations take the same actions at all reachable states, and can differ only on unreachable states.

that can be run by all vehicles, including those without V2V communications equipment. One example of such σ is the traffic light policy σ^{TL}. In all cases, moves permitted by this policy will have priority, but we allow vehicles to violate the policy provided that they know that they can do so safely. To avoid clashes, we establish a priority order on the violations. Let *next* be a function from histories such that $\mathit{next}(h) \in \mathcal{L}_{in}$ for each history h. Intuitively, the agent at the front of the queue for lane $\mathit{next}(h)$ will get precedence in going through the intersection at the point (r, m). The context γ is *next-aware* if, for all protocols P for γ and agents i and $\ell \in \mathcal{L}_{in}$, we have that $\mathcal{I}_{\gamma,P} \models \mathit{next} = \ell \Rightarrow K_i(\mathit{next} = \ell)$.

Consider the following knowledge-based program $\mathbf{P}$, where V_i is the proposition

> $(\mathit{lane}_i, \mathit{intent}_i) \notin \sigma$ and the move $(\mathit{lane}_i, \mathit{intent}_i)$ is compatible with (a) all moves $(\mathit{lane}_j, \mathit{intent}_j) \in \sigma$ where j is an agent who is about to enter the intersection (i.e., going_j holds) (b) all moves $(\mathit{lane}_j, \mathit{intent}_j) \notin \sigma$ where $j \neq i$ is an agent for which going_j holds and $\mathit{lane}_j \in [\mathit{next}, \mathit{lane}_i)$. (Here $[\mathit{next}, \mathit{lane}_i)$ is the set of lanes from $\mathit{next}(r, m)$ to lane_i (mod $|\mathcal{L}_{in}|$).)

Program $\mathbf{P}_i$.

```
if $K_i(\mathit{front}_i \wedge ((\mathit{lane}_i, \mathit{intent}_i) \in \sigma \vee V_i))$ then go
else noop
```

Intuitively, this knowledge-based program allows all agents permitted by σ to go to do so, as well as allowing agents not permitted by σ to go, provided they do so in a cyclic priority order, and each agent that goes knows that its move is compatible with the moves of all agents of higher priority (including agents permitted to go by σ).

▶ **Proposition 10.** *Let σ be a conflict-free intersection policy. If context γ is synchronous, next-aware, and σ-aware, then there exists a unique implementation P of $\mathbf{P}$ that satisfies safety and validity, is lexicographically optimal with respect to γ, and lexicographically dominates the unique implementation of $\mathbf{P}^{\sigma}$. Moreover, if σ is fair then P satisfies liveness.*

We can also obtain liveness of the implementations of $\mathbf{P}$ under some other conditions. Define the function *next* to be *fair* if, for all feasible sequences of histories $h_0, h_1, \ldots$, all $m \geq 0$ and all lanes $\ell \in \mathcal{L}_{in}$, there exists $m' \geq m$ such that $\mathit{next}(h_{m'}) = \ell$. Intuitively, fairness of *next* will ensure that *next* fairly selects the first agent that can violate the intersection policy according to $\mathbf{P}$ when this can be done safely.

We also need to ensure that it is not the case that σ always gives priority to other lanes whenever the lane ℓ is selected by *next*. For this, define a pair (σ, next), consisting of an intersection policy σ and a function *next*, to be *fair* if for all feasible sequences of histories $h_0, h_1, \ldots$, all $m \geq 0$ and all moves $(\ell, \ell') \in \mathcal{L}_{in}$, there exists $m' \geq m$ such that either $(\ell, \ell') \in \sigma(h_{m'})$, or $\mathit{next}(h_{m'}) = \ell$ and (ℓ, ℓ') is compatible with all the moves in $\sigma(h_{m'})$

▶ **Proposition 10.** *Let P be an implementation of $\mathbf{P}$ with respect to a synchronous, next-aware and σ-aware context. If the pair (σ, next) is fair, then P satisfies liveness.*

Note that if *next* is fair, and the $\sigma_\emptyset$ is the (unfair) intersection policy defined by $\sigma_\emptyset(h) = \emptyset$ for all histories h, then the pair $(\sigma_\emptyset, \mathit{next})$ is fair. For examples in which σ is not trivial, consider the following properties of σ. Say that σ is *cyclic* (with cycle length k) if for all histories h and h' with $|h| \equiv |h'| \mod k$, we have $\sigma(h) = \sigma(h')$. Say that σ is *non-excluding* if for all moves (ℓ, ℓ'), there exists a history h such that (ℓ, ℓ') is compatible with all moves in $\sigma(h)$. Given a non-excluding σ with cycle length k, let *next* be defined by $\mathit{next}(h) = \lfloor h/k \rfloor \mod k$. Then (σ, next) is fair. This is because the value of *next* cycles through all values in

$\mathcal{L}_{in}$, but is held constant through each cycle of σ. Thus, for each move (ℓ, ℓ'), eventually a point in these combined cycles will be reached for which the value of *next* is ℓ and (ℓ, ℓ') is compatible with all moves permitted by σ.

8.1 Implementing $\mathbf{P}$ when there is no communication

We now consider standard implementations of $\mathbf{P}$ in two particular contexts of interest. Since we would like the implementations to be correct and lexicographically optimal, we use *next* and σ defined as $\mathit{next}(h) = m \mod |\mathcal{L}_{in}|$ and $\sigma(h) = \emptyset$ for all histories h of length m. Using this choice of *next* and σ in the construction of $\mathbf{P}$ ensures that in any synchronous intersection context, both *next-awareness* and σ*-awareness* hold; moreover, the pair (next, σ) is fair. Therefore, implementations P of $\mathbf{P}$ in such contexts are correct and lexicographically optimal, by Propositions 10 and 10.

We have taken σ to be empty for ease of exposition. For practical implementations, the construction given by the proof of Proposition 10 can be used to get other implementations that prioritize moves permitted by σ. (For example, in an intersection where certain lanes are often busier, moves originating from those lanes can be prioritized.) Note that for empty σ, the condition $K_i(\mathit{front}_i \wedge ((\mathit{lane}_i, \mathit{intent}_i) \in \sigma \vee V_i))$ reduces to $K_i(\mathit{front}_i \wedge V_i')$, where V_i' is the proposition

> "the move $(\mathit{lane}_i, \mathit{intent}_i)$ is compatible with all moves $(\mathit{lane}_j, \mathit{intent}_j)$ of agents $j \neq i$ with $\mathit{lane}_j \in [\mathit{next}, \mathit{lane}_i)$ such that going_j",

since σ is empty. Consider the following context with no communication. Let $\gamma_\emptyset$ be a synchronous intersection context where agents do not broadcast messages. Formally, for a failure model $\mathcal{F}$, we define $\gamma_\emptyset(\mathcal{F}) = (\mathcal{E}_\emptyset, \mathcal{F}, \pi_\emptyset)$, where

- $(\mathcal{E}_\emptyset)_i$ is an information-exchange protocol where the following hold:
  - The set of memory states is a singleton so, effectively, local states consist only of the sensor reading $L_i = \Sigma_i$.
  - No messages are sent, so $M_i = \emptyset$, μ_i is the constant function with value $\perp$, and δ_i is omitted.
  - The sensor model is defined as in the definition of intersection contexts. The only modification is that the sensor model now maps environment states to tuples of the form $\langle \mathit{front}_i, \mathit{lane}_i, \mathit{intent}_i, \mathit{time}_i \rangle$, where time_i is determined by the time encoded in the environment state.
- $\pi_\emptyset$ interprets the propositions defined for intersection contexts in the obvious way.

We now define a procedure to compute a set Pos_i of moves that agent i believes may be performed as a function of *next* and the structure of the intersection represented by $\mathcal{O}$. We capture stages of the construction of this set as sets of moves Pos_I^l for $l \in [\mathit{next}-1, \mathit{lane}_i)$. (By *next*-awareness, *next* is computable from the agent's local state. For brevity, we interpret $\mathit{next}-1$ as $\mathit{next}-1(\mod |\mathcal{L}_{in}|)$.)

1. Start with $\mathit{Pos}_i = \mathit{Pos}_i^{\mathit{next}-1} = \emptyset$
2. For $l \in [\mathit{next}, \mathit{lane}_i)$ do
   a. Let L be the set of moves (l, l') where $l' \in \mathcal{L}_{out}$ such that (l, l') is compatible with Pos_i, and let $\mathit{Pos}_i^l := \mathit{Pos}_i \cup L$ and $\mathit{Pos}_i := \mathit{Pos}_i^l$.
3. Output Pos_i.

Let $P^\emptyset$ be the standard protocol given by the following program, where move (l, l') is compatible with a set of moves S if it is compatible with all moves in S according to $\mathcal{O}$.

Program $P_i^{\emptyset}$.

```
if $front_i \wedge (lane_i, intent_i)$ is compatible with $Pos_i$ then go
else noop
```

▶ **Proposition 10.** *$P^{\emptyset}$ implements $\mathbf{P}$ with respect to $\gamma_{\emptyset}(\mathcal{F})$ for $\mathcal{F} \in \{NF, CR, SO\}$.*

Proposition 10 shows that, without communication, a protocol that essentially implements traffic lights is lexicographically optimal.

8.2 Implementing P in a context with limited communication

If we allow messages regarding the current lane and agents' intentions by agents that reach the front, this changes how implementations of $\mathbf{P}$ behave. Roughly speaking, in runs where the intersection gets crowded, a much larger set of agents can proceed through the intersection. Let γ_{intent} be a synchronous context with communication failures such that if an agent is in the front of some lane, it broadcasts $(lane, intent)$. (This information exchange broadcasts a lot less information than a full-information exchange.) More formally,[3] for a failure model $\mathcal{F}$, we define $\gamma_{intent}(\mathcal{F}) = (\mathcal{E}_{intent}, \mathcal{F}, \pi_{intent})$, where

- $(\mathcal{E}_{intent})_i$ is defined as an information-exchange protocol where the following hold:
  - The local states maintain a set of moves M in the memory component in addition to the sensor readings. Intuitively, this set represents the set of moves from broadcasts that were received by i in the current round. Note that M_i may not contain i's move since i's broadcast may fail.
  - The set of messages is $M_i = \mathcal{L}_{in} \times \mathcal{L}_{out}$, and μ_i broadcasts the message $(lane_i, intent_i)$ by reading $lane_i$ and $intent_i$ from the sensor reading, if $front_i$, and broadcasts no message otherwise. Note that these variable references are from $\mathscr{S}(s'_e)$ where s'_e is the new environment state that the system moves to in the course of the round.
  - The sensor model is defined as in the definition of intersection contexts (while including $time$ as a sensor reading as in $\mathcal{E}_{\emptyset}$).
  - δ_i maps the set of received messages directly into the memory with i's own move; that is, $\delta_i(s_i, a, Mes) = Mes$. Note that an agent can determine from this set whether its own broadcast was successful.
- π_{intent} interprets the propositions defined for interpretation contexts in the obvious way.

We now proceed as in Subsection 8.1 and define a procedure to compute from an agent i's local state $s_i = (M_i, (lane_i, intent_i, time_i))$ a set Pos_i of moves that agent i believes may be performed by higher-priority agents in the next round. We again capture stages of the construction of this set as sets of moves Pos_I^l for $l \in [next - 1, lane_i)$.

```
1. Start with $Pos_i = Pos_i^{next-1} = \emptyset$
2. For $l \in [next, lane_i)$ do
   a. If for some $l' \in \mathcal{L}_{out}$, the move $(l, l')$ is in $M_i$ then
        if $(l, l')$ is compatible with $Pos_i$
        then $Pos_i^l := Pos_i \cup \{(l, l')\}$ and $Pos_i := Pos_i^l$
        else $Pos_i^l := Pos_i$.
```

[3] This context satisfies the *sufficiently rich* condition of Section 6.

b. Otherwise, let L be the set of moves (l, l') where $l' \in \mathcal{L}_{out}$ such that (l, l') is compatible with Pos_i, and let $Pos_i^l := Pos_i \cup L$ and $Pos_i := Pos_i^l$.[4]

3. Output Pos_i.

Let the output of running this procedure on a local state with memory state M_i be denoted by Pos_i, and let P^{intent} be the standard protocol defined using the following program:

Program P_i^{intent}.

```
if $front_i \wedge (lane_i, intent_i)$ is compatible with $Pos_i$ then go
else noop
```

▶ **Proposition 10.** *P^{intent} implements $\mathbf{P}$ with respect to $\gamma_{intent}(\mathcal{F})$ for $\mathcal{F} \in \{CR, SO\}$.*

Again, by Proposition 10, it follows that the intersection protocol P^{intent} is lexicographically optimal with respect to the contexts $\gamma_{intent}(\mathcal{F})$ for for $\mathcal{F} \in \{CR, SO\}$.

9 Discussion

We introduced the *intersection problem*, identified the appropriate notion of optimality called *lexicographical optimality*, and designed protocols that are optimal in a variety of contexts. A knowledge-based analysis and the use of *intersection policies* were crucial in this process.

Previous work has considered many models ranging from computing individual trajectories of vehicles to relying on centralized schedulers [6]. In [16, 15], a four-way intersection is considered in a context with failures. [10, 17] consider *virtual traffic lights*; the approach is evaluated using a large-scale simulation. [9] solves the same problem probabilistically, in contexts with failures. Work in the control theory literature has focused on vehicle dynamics when going through an intersection [11] to avoid collision. Efforts have also been made to build distributed intersection management systems through V2V communication [5].

While there has been considerable effort in designing protocols for specific intersections or designing architectures for intersection management systems, we aim to develop a context- and architecture-independent approach. Our goal in this paper is to lay the theoretical foundations of optimal intersection protocol design in a variety of contexts, including contexts with failures. We do so abstractly by defining the model to capture any intersection topology with minimal requirements on V2V communication range. While the protocols we design do not require sensors such as lidar and radar, the use of a knowledge-based program $\mathbf{P}$ provides a direct method to develop optimal implementations in contexts with extra sensors.

The problem we study in this paper can be viewed as a generalization of the classical problem of *mutual exclusion*, which requires that two distinct agents are not simultaneously in a *critical section* of their code. Indeed, a variant of mutual exclusion called *group mutual exclusion* [12] is strictly weaker than the intersection problem. In group mutual exclusion, each process is assigned a session when entering the critical section and processes are allowed to enter the critical section simultaneously provided that they share the same session. If agents form an equivalence relation based on their move compatibility according to $\mathcal{O}$, we can identify each equivalence class to be in the same session and think of the intersection as the critical section. However, our setting differs in some critical ways:

[4] Intuitively, since M is the set of moves that i hears about from agents in the front of some lane, in this case i did not hear from anyone in lane l. However, in settings with sending omissions, there may nevertheless be an agent at the front of lane l''. Such an agent will move only if it can do so safely.

- Intersections often have an $\mathcal{O}$ relation that is not an equivalence relation. For instance, the fact that agents' moves conflict in lanes A-B and in lanes B-C does not imply that their moves in lanes A and C conflict (e.g., if agents want to move straight in a four-way intersection with two lanes in each direction).
- We take the set *Ag* of agents to be unbounded, while group mutual exclusion (and equivalent problems such as *room synchronization* [3]) consider a bounded number of agents.
- Our agents arrive according to a (possibly infinite) schedule determined by the adversary.
- To the best of our knowledge, fault-tolerance has not been considered in the group mutual-exclusion setting.

The mutual-exclusion problem is generally studied with respect to an interleaving model of asynchronous computation, but as Lamport [13] noted, this model is not physically realistic, and already builds in a notion of mutual exclusion between the actions of distinct agents. The *Bakery* mutual-exclusion protocol [13] is correct with respect to models allowing simultaneous read and write operations. Moses and Patkin [14] develop an improvement of Lamport's Bakery algorithm for the mutual-exclusion problem using a knowledge-based analysis, noting that there are situations in which Lamport's protocol could enter the critical section, but fails to do so. A weaker knowledge-based condition for mutual exclusion is used by Bonollo et al. [4]; it states that an agent i may enter its critical section when it knows that no other agent will enter its critical section until agent i has exited from its critical section. Clearly these knowledge-based approaches are similar in spirit to ours. We hope to study the exact relationship between these problems in the near future.

There are several directions that we hope to explore in the future. One involves extending the current results to contexts with stronger adversaries and evaluating implementations of **P** in other contexts. Another is considering strategic agents, who may deviate from a protocol to cross the intersection earlier.

References

1 K. Alpturer, J. Y. Halpern, and R. van der Meyden. Optimal eventual Byzantine agreement protocols with omission failures. In *Proc. 42nd ACM Symposium on Principles of Distributed Computing*, pages 244–252, 2023.

2 K. Alpturer, J. Y. Halpern, and R. van der Meyden. A knowledge-based analysis of intersection protocols, 2024. arXiv:2408.09499.

3 Guy E. Blelloch, Perry Cheng, and Phillip B. Gibbons. Room synchronizations. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2001, Heraklion, Crete Island, Greece, July 4-6, 2001*, SPAA '01, pages 122–133, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/378580.378605.

4 U. Bonollo, R. van der Meyden, and E.A. Sonenberg. Knowledge-based specification: Investigating distributed mutual exclusion. In *Bar Ilan Symposium on Foundations of AI*, 2001. URL: https://www.cse.unsw.edu.au/~meyden/research/bisfai.pdf.

5 António Casimiro, Jörg Kaiser, Elad M. Schiller, Pedro Costa, José Parizi, Rolf Johansson, and Renato Librino. The karyon project: Predictable and safe coordination in cooperative vehicular systems. In *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 1–12, 2013. doi:10.1109/DSNW.2013.6615530.

6 Lei Chen and Cristofer Englund. Cooperative intersection management: A survey. *Trans. Intell. Transport. Syst.*, 17(2):570–586, January 2016. doi:10.1109/TITS.2015.2471812.

7 K. Dresner and P. Stone. A multiagent approach to autonomous intersection management. *Journal of A.I. Research*, 31:591–656, 2008. doi:10.1613/JAIR.2502.

8 R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge.* MIT Press, Cambridge, MA, 1995. A slightly revised paperback version was published in 2003.

9 N. Fathollahnejad, E. Villani, R. Pathan, R. Barbosa, and J. Karlsson. On reliability analysis of leader election protocols for virtual traffic lights. In *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 1–12, 2013.

10 M. Ferreira, R. Fernandes, H. Conceição, W. Viriyasitavat, and O. K. Tonguz. Self-organized traffic control. In *Proceedings of the Seventh ACM International Workshop on VehiculAr InterNETworking*, pages 85–90, 2010.

11 Michael R. Hafner, Drew Cunningham, Lorenzo Caminiti, and Domitilla Del Vecchio. Cooperative collision avoidance at intersections: Algorithms and experiments. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1162–1175, 2013. doi:10.1109/TITS.2013.2252901.

12 Yuh-Jzer Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, November 2000. doi:10.1007/PL00008918.

13 Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974. doi:10.1145/361082.361093.

14 Yoram Moses and Katia Patkin. Mutual exclusion as a matter of priority. *Theor. Comput. Sci.*, 751:46–60, 2018. doi:10.1016/j.tcs.2016.12.015.

15 E. Regnath, M. Birkner, and S. Steinhorst. CISCAV: consensus-based intersection scheduling for connected autonomous vehicles. In *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*, pages 1–7, 2021.

16 V. Savic, E. M. Schiller, and M. Papatriantafilou. Distributed algorithm for collision avoidance at road intersections in the presence of communication failures. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1005–1012, 2017.

17 Rusheng Zhang, Frank Schmutz, Kyle Gerard, Aurélicn Pomini, Louis Basseto, Sami Ben Hassen, Akihiro Ishikawa, Inci Ozgunes, and Ozan Tonguz. Virtual traffic lights: System design and implementation. In *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*, pages 1–5, 2018. doi:10.1109/VTCFall.2018.8690709.

Byzantine Resilient Distributed Computing on External Data

John Augustine ✉ 🏠
Indian Institute of Technology Madras, Chennai, India

Jeffin Biju ✉
Indian Institute of Technology Madras, Chennai, India

Shachar Meir ✉
Weizmann Institute of Science, Rehovot, Israel

David Peleg ✉ 🏠
Weizmann Institute of Science, Rehovot, Israel

Srikkanth Ramachandran ✉
Indian Institute of Technology Madras, Chennai, India

Aishwarya Thiruvengadam ✉ 🏠
Indian Institute of Technology Madras, Chennai, India

Abstract

We study a class of problems we call *retrieval* problems in which a distributed network has read-only access to a trusted external data source through queries, and each peer is required to output some computable function of the data. To formalize this, we propose the *Data Retrieval Model* comprising two parts: (1) a congested clique network with k peers, up to βk of which can be Byzantine in every execution (for suitable values of $\beta \in [0, 1)$); (2) a trusted source of data with no computational abilities, called the *External Data Source* (or just source for short). This source stores an array $\mathcal{X}$ of n bits ($n \gg k$), providing every peer in the congested clique read-only access to $\mathcal{X}$ through queries. It is assumed that a query to the source is significantly more expensive than a message between two peers in the network. Hence, we prioritize minimizing the number of queries a peer performs over the number of messages it sends. Retrieval problems are easily solved by having each peer query all of $\mathcal{X}$, so we focus on designing non-trivial query-efficient protocols for retrieval problems in the DR network that achieve low query performance per peer. Specifically, to initiate this study, we present deterministic and randomized upper and lower bounds for two fundamental problems. The first is the Download problem that requires every peer to output an array of n bits identical to $\mathcal{X}$. The second problem of focus, Disjunction, requires nodes to learn if some bit in $\mathcal{X}$ is set to 1.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases Byzantine Fault Tolerance, Blockchain Oracle, Congested Clique, Data Retrieval Model

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.3

Related Version *Full Version*: `https://arxiv.org/abs/2309.16359`

Funding *John Augustine*: Supported by the Cybersecurity Centre, IIT Madras.
David Peleg: Venky Harinarayanan and Anand Rajaraman Visiting Chair Professor. The funds from this professorship enabled exchange visits between IIT Madras, India, and the Weizmann Institute of Science, Israel.

Acknowledgements We would like to thank Atharva Chougule for useful ideas.

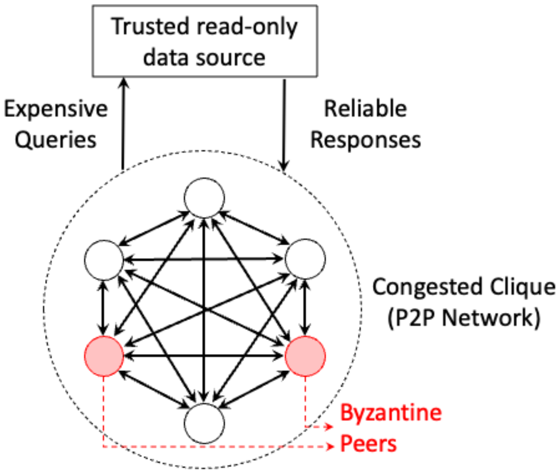
38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 3; pp. 3:1–3:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Background and Motivation

We study distributed systems in which a peer-to-peer (P2P) network retrieves data (or some boolean function of it) from a trusted source of data that is external to the network. To formalize this study, we propose a new model called the *Data Retrieval (DR) Model* comprising a congested clique network and an *External Data Source* (or source for short) with no computational capabilities. The DR model consists of a congested clique network with k peers, up to βk of which can be Byzantine in every execution (for suitable values of $\beta \in [0, 1)$). The source comprises an array $\mathcal{X}$ of n bits ($n \gg k$), providing every peer in the congested clique read-only access to $\mathcal{X}$ through queries. We prioritize minimizing the number of queries a peer performs over the number of messages it sends as we assume that a query to the source is significantly more expensive than a message between two peers in the network.

Our DR model is inspired by distributed Blockchain oracles [7, 12]. In such oracle systems, a decentralized P2P network with some Byzantine corruptions (modeled by our congested clique network) is tasked with retrieving information from trusted external data sources (e.g., stock prices, inflation indices, IoT sensors, etc.) through well defined *Application Programming Interface* (API) calls. Currently, nodes in state-of-the-art blockchain oracles do not cooperate, resulting in each node having to read all the information directly from the data source. These API calls can be expensive with cost scaling directly with their usage. The DR model provides a framework for designing Byzantine resilient mechanisms for nodes in such P2P networks to share the workload of queries, thus reducing the cost for each node.



Difference in approach from traditional BFT problems. The theory of Byzantine fault tolerance has been a fundamental part of distributed computing ever since its introduction by Pease, Shostak, and Lamport [33, 36] in the early 80's, and has had a profound influence on cryptocurrencies, blockchains, distributed ledgers, and other decentralized peer-to-peer systems. It largely focused on a canonical set of problems like Broadcast [18], Agreement [11, 33, 36, 37], k-set Agreement [14], Common Coin [34], and State Machine Replication [13]. Some studies have injected Byzantine fault tolerance into other related areas (cf. [5, 6, 9, 16, 17]). In most of these studies, the main parameter of interest is the maximum fraction β of the peers that can be corrupted by the adversary in an execution.

Consider the Byzantine Agreement problem that requires n peers, each with an input bit, to agree on a common output bit that is *valid*, in the sense that at least one honest (non-Byzantine) peer held it as input. In the synchronous setting, even without cryptographic assumptions, there are agreement algorithms that can tolerate any fraction $\beta < 1/3$ of Byzantine peers [33] (and this extends to asynchronous settings as well [11]). When $\beta \geq 1/3$, agreement becomes impossible in these settings [33]. However, the bound improves to $\beta < 1/2$ with message authentication by cryptographic digital signatures [38]. By the well-known network partitioning argument (discussed shortly), $\beta < 1/2$ is required for any form of Byzantine agreement. For most of the Byzantine fault tolerance literature, β hovers around either 1/3 or 1/2, with some notable exceptions like authenticated broadcast [18] that can tolerate any $\beta < 1$.

The main reason for this limitation stems from the inherent coupling of data and computing. Consider, for instance, any Byzantine Agreement variation with $\beta \geq 1/2$. When all honest peers have the same input bit (say, 1), the Byzantine peers hold at least half the input bits and can unanimously claim 0 as their input bits. This ability of Byzantine peers to spoof input bits makes it fundamentally impossible for honest peers to reach a correct agreement with the validity requirement intact. At the heart of this impossibility is the adversary's power to control information crucial to solving the problem. In fact, this issue leads to many impossibilities and inability to solve problems exactly (see e.g, [4]).

In contrast, having a reliable source that provides the data in read-only fashion yields a distributed computing context where access to data cannot be controlled by Byzantine peers. Taken to the extreme, any honest peer can individually solve all problems by directly querying the source for all required data. However, queries are charged for, and can be quite expensive. So the challenge is to design effective and secure collaborative techniques to solve the problem at hand while minimizing the number of queries made by each honest peer[1]. Hence, despite the source being passive (read-only with no computational power), its reliability makes the model stronger than the common Byzantine model.

The Model

A Data Retrieval model consists of (i) k peers that form a *congested clique* and (ii) a source of data that is external to the congested clique called the *source* that stores the input array comprising n bits and provides read-only access to its content through queries.

Congested Clique. The k peers are identified by unique ID's assumed to be from the range $[1, k]$. The peers are connected via a complete network. In each round, every peer can send at most one $O(\log n)$ bit message to each of the other peers. This communication mechanism is referred to as *peer-peer* communication.

The source. The n-bit input array $\mathcal{X} = \{x_1, \ldots, x_n\}$ (with $n \gg k$) is stored in the source. It allows peers to retrieve that data through queries of the form $\texttt{Query}(i)$, for $1 \leq i \leq n$. The answer returned by the source would then be x_i, the i^{th} element in the array. This type of communication is referred to as *source-peer* communication.

[1] Note that appointing some individual peers to query each input bit and applying a *Byzantine Reliable Broadcast (BRB)* protocol [2, 11, 18] for disseminating the bits to all peers will not do, since the appointed peers might be Byzantine, in which case the BRB protocol can only guarantee agreement on *some* value, but not necessarily the true one. Moreover, Byzantine Reliable Broadcast (BRB) cannot be solved when $\beta \geq 1/3$ with no authenticated messages.

Synchrony and rounds. We consider a synchronous round setting where peers share a global clock, and the network delay is bounded by Δ. Each round has a total length of 3Δ and consists of two sub-rounds:

1. The *query sub-round* of length 2Δ of source-peer communication, comprising sending queries of the form `Query`$(\cdot)$ from a peer to the source and receiving the responses from the source. Every peer can send up to n queries per round to the source. (This is merely an upper limit; our protocols typically send significantly fewer queries).
2. The *message-passing sub-round* of length Δ of peer-peer communication, consisting of messages exchanged between peers. Every message is of size $O(\log n)$

We assume local computation takes 0 time and is performed at the beginning of a round. We assume that a peer M can choose to ignore (not process) messages received from another peer during the execution. Such messages incur no communication cost[2] for M.

The adversarial settings. The behavior of the environment in which our protocols operate is modeled via an adversary `Adv` that is in charge of selecting the input data and fixing the peers' failure pattern. In executing a protocol, a peer is considered *honest* if it obeys the protocol throughout the execution. A *Byzantine* peer can deviate from the protocol arbitrarily (controlled by `Adv`). The adversary `Adv` can corrupt at most βk peers for some given[3] $\beta \in [0, 1)$. This implies that `Adv` cannot corrupt all of the peers; our results are stated under this assumption. Letting $\gamma = 1 - \beta$, there is (at least) a γ fraction of honest peers. We denote the set of Byzantine (respectively, honest) peers in the execution by $\mathcal{B}$. (resp., $\mathcal{H}$).

We design both deterministic and randomized protocols. When the protocol is deterministic, the adversary can be thought of as all-knowing. Thus, `Adv` knows exactly how the complete execution will proceed and can select Byzantine nodes from the beginning based on this knowledge. When the protocol is randomized, the peers may generate random bits locally. At the beginning of each round i, `Adv` has knowledge of $\mathcal{X}$, all the local random bits generated up to round $i - 1$, and all peer-peer and source-peer communications up to round $i - 1$. At the start of round i, it can corrupt as many peers as it desires, provided the total number of peers corrupted since the beginning of the execution does not exceed βk. Such an adversary is said to be *adaptive*.

Complexity measures. The following complexity measures are used to analyze our protocols: (i) *Query* Complexity ($\mathcal{Q}$): the maximum number of queries made by an honest peer during the execution of the protocol, (ii) *Message* Complexity ($\mathcal{M}$): the total number of messages sent by honest peers during the execution of the protocol, and (iii) *Round* Complexity ($\mathcal{T}$): the number of rounds (or *time*) it takes for the protocol to terminate.

As queries to the source are expected to be the more expensive component in the foreseeable future, we primarily focus on optimizing the query complexity $\mathcal{Q}$, only trying to optimize $\mathcal{T}$ and $\mathcal{M}$ when $\mathcal{Q}$ is optimal (within $\log(n)$ factors). Our definition of $\mathcal{Q}$ (measuring the maximum cost per peer rather than the total cost) favors a fair and balanced load of queries across honest peers.

[2] Specifically, an honest peer M can ignore the messages of a known Byzantine peer M' and thus thwart any "denial of service" attack that M' attempts on M. Such messages sent by the Byzantine peer M' to M will not be counted towards the message complexity.

[3] We do not assume β to be a fixed constant (unless mentioned otherwise).

Problems Studied and Their Complexity in the Failure Free Model

We introduce the two main problems we focus on in this paper. To establish a baseline for our various results, we first outline the best possible complexity measures when there are no Byzantine failures. For $\mathcal{Q}$, the best bound is the total number of queries required divided by k, since this work of querying can be distributed evenly.

Download. We begin with the fundamental Download problem, where each of the k peers needs to obtain a copy of all n input bits from the cloud. This problem is the most fundamental retrieval problem since every computable function f of the input can be computed by the peers by first running a download protocol and then computing $f(\mathcal{X})$ locally at no additional costs. Hence, its query cost serves as a baseline against which to compare the costs of other specialized algorithms for specific problems. Observe that a $\mathcal{Q}$ lower bound for computing any Boolean function on $\mathcal{X}$ serves as a lower bound for Download as well.

To solve this problem in the absence of failures, all n bits need to be queried, and this workload can be shared evenly among k peers, giving $\mathcal{Q} = \Theta(n/k)$. The message complexity is $\mathcal{M} = \tilde{O}(nk)$ and round complexity is $\mathcal{T} = \tilde{O}(n/k)$ since $\Omega(n/k)$ bits need to be sent along each communication link when the workload is shared.

Disjunction. In the Disjunction problem, the honest peers must learn whether at least one of the input bits in $\mathcal{X}$ is a 1. We also consider an *Explicit* Disjunction version where each peer must learn an index i such that $\mathcal{X}[i] = 1$ (or output 0 if there are no 1's).

The Disjunction problem is a retrieval problem that illustrates the possibility of achieving better results than trivially using Download as a subroutine. The complexity of the problem is closely tied to the *density* δ (i.e., the fraction of ones) in the input. In fact, the relevant parameter is often $1/\boldsymbol{\delta}$ where $\boldsymbol{\delta} = \max(1/n, \delta)$ to handle the exceptional case when $\delta = 0$.

Let us consider the Explicit Disjunction problem. In the deterministic setting, at least $n - \boldsymbol{\delta}n + 1$ queries are required in total. Consequently, the best deterministic query complexity is $\mathcal{Q} = O(n(1 - \boldsymbol{\delta})/k)$. The round complexity is $\mathcal{T} = O(1)$, and message complexity is $\mathcal{M} = O(k)$. Peers that find a 1-bit can send the index to a "leader" peer to broadcast the answer.

Randomization helps when δ is large. Querying $\left(\boldsymbol{\delta}^{-1} \cdot \ln n\right)$ bits uniformly at random in search for a 1 bit has failure probability of $(1 - \boldsymbol{\delta})^{\ln n/\boldsymbol{\delta}} \leq 1/n$. Thus $O\left(\boldsymbol{\delta}^{-1} \cdot \ln n\right)$ queries are sufficient to find a 1 w.h.p. Even without knowledge of δ, one can simply try density values in decreasing powers of 2, starting with 1/2 and eventually land at a 1 having made at most $O\left(\boldsymbol{\delta}^{-1} \cdot \ln n\right)$ queries. We can distribute the work equally amongst k peers, and thus $\mathcal{Q} = O(1 + \boldsymbol{\delta}^{-1} \cdot \frac{1}{k} \cdot \ln n)$. The time and message analysis is similar to the deterministic case, i.e, $\mathcal{T} = O(1)$, $\mathcal{M} = O(k)$. Note that $\mathcal{Q} = \Omega(\frac{1}{k} \cdot \boldsymbol{\delta}^{-1})$, for any algorithm that solves the Disjunction problem with constant probability.

Our Contributions

We initiate the study of the Data Retrieval Model and retrieval problems. We present several deterministic and randomized protocols and some lower bounds for Download and Disjunction. Here, we state only simplified bounds, in which the $\tilde{O}(\cdot)$ notation hides factors dependent on β and poly log factors in n. The main results are summarized in Table 1 for convenience.

Download. For the deterministic model, the Download problem turns out to be expensive, requiring $\Omega(\beta n)$ queries in the worst case. Every peer essentially has to query the entire input array for itself. In the randomized model, we give an algorithm that solves the Download problem (and consequently *any* function of the input) for an *arbitrary* fraction $\beta < 1$ of Byzantine faults while requiring at most $\tilde{O}(n/k + \sqrt{n})$ queries per peer. The result is nearly as efficient as the *failure-free model* whenever $k < \sqrt{n}$. The time and message costs are $\mathcal{T} = O(n)$ and $\mathcal{M} = \tilde{O}(kn + k^2\sqrt{n})$. A natural question then, is whether the additive $\sqrt{n}$ term is necessary for $k > \sqrt{n}$. While we are not able to fully address this question, we show that for restricted β ($< 1/3$), we can be fully efficient for all $k \in [1, n]$, getting $\mathcal{Q} = \tilde{O}\left(\frac{n}{k}\right)$, $\mathcal{T} = \tilde{O}(n)$, and $\mathcal{M} = \tilde{O}(nk^2)$.

Disjunction. To show that for specific problems one can be more efficient, we consider Disjunction when the input bits have density δ. Naturally, the problem becomes easier as δ gets larger. We first show that any deterministic algorithm requires $\Omega(n/k + \boldsymbol{\delta}^{-1})$ queries in the worst case. Next, we show that for any $\beta < 1$, there exists a deterministic algorithm that makes $\tilde{O}(n/k + \boldsymbol{\delta}^{-1} + k)$ queries. This algorithm is nearly optimal whenever $k < \sqrt{n}$. Our second deterministic algorithm achieves near optimal complexity provided $\beta < 1/2$. Both algorithms require $\mathcal{T} = \tilde{O}(1)$ and $\mathcal{M} = \tilde{O}(k^2)$.

We then consider the randomized model. It is easy to see that any algorithm requires $\Omega(1/k \cdot \boldsymbol{\delta}^{-1})$ queries per peer. We show that this is nearly tight by presenting an algorithm that w.h.p. solves the Disjunction problem with $\mathcal{Q} = \tilde{O}\left(\frac{1}{k} \cdot \boldsymbol{\delta}^{-1}\right)$, $\mathcal{T} = \tilde{O}\left(1\right)$, $\mathcal{M} = \tilde{O}(k^2)$.

Table 1 Our Main Results (with β treated as constant).

Problem & Model	Query	Lower Bound	Round	Message	Theorem
Download					
Randomized $\beta < 1$	$\tilde{O}(n/k + \sqrt{n})$	$\Omega(n/k)$	$O(n)$	$\tilde{O}(nk + k^2\sqrt{n})$	Thm 4
Randomized $\beta < 1/3$	$\tilde{O}(n/k)$	$\Omega(n/k)$	$O(n)$	$\tilde{O}(nk^2)$	Thm 12
Disjunction					
Deterministic $\beta < 1$	$\tilde{O}(n/k + \boldsymbol{\delta}^{-1} + k)$	$\Omega\left(\boldsymbol{\delta}^{-1} + \frac{(1-\delta)n}{\gamma k}\right)$	$\tilde{O}(1)$	$\tilde{O}(k^2)$	Thm 17
Deterministic $\beta < 1/2$	$\tilde{O}(n/k + \boldsymbol{\delta}^{-1})$	$\Omega\left(\boldsymbol{\delta}^{-1} + \frac{(1-\delta)n}{\gamma k}\right)$	$\tilde{O}(1)$	$\tilde{O}(k^2)$	Thm 18
Randomized $\beta < 1$	$\tilde{O}(1/k \cdot \boldsymbol{\delta}^{-1})$	$\Omega(\frac{1}{\gamma k} \cdot \boldsymbol{\delta}^{-1})$	$\tilde{O}(1)$	$\tilde{O}(k^2)$	Thm 19

2 Methods

Private ρ-Representative Committees. Several of our protocols organize the peers in *committees*, assigned to perform a common task. In a *private ρ-representative committee*, every peer knows only whether it belongs to the committee and the committee is guaranteed to have at least ρ honest members, where ρ is known.

We present a probabilistic construction for a ρ-representative committee, where the guarantee of at least ρ honest members holds w.h.p. To construct such a committee, each peer adds itself to the committee with probability p. See Algorithm 1. By choosing an appropriate value of p, we can obtain high probability guarantees on the number of (honest) peers in a committee using standard Chernoff tail bounds. This yields the following result

▶ **Lemma 1.** *Consider k i.i.d Bernoulli random variables with bias $p = \min(1, \frac{9\ln n + 4\rho}{\gamma k})$, $\beta \in [0, 1)$, $n > 1$ and $\rho \le \gamma k$, we have with probability at least $1 - 2n^{-3}$,*

- *for any subset of γk variables, at least ρ of them are 1.*
- *At most $(18 \ln n + 8\rho)/\gamma$ variables are 1.*

Lemma 1 implies that w.h.p a committee $\mathcal{C}$ constructed by Algorithm 1 is indeed a private ρ-representative committee and it will have at most $(18 \ln n + 8\rho)/\gamma$ honest members.

Algorithm 1 Procedure `Elect_Private`.

```
Every peer tosses a biased coin with a probability of heads $p = \min\left\{\frac{6\ln n + 4\rho}{\gamma k}, 1\right\}$
return $\mathcal{C}$ = set of peers that tossed heads.
```

Commit Verification. Before a peer M commits b as x_i, it verifies that $x_i = b$ by one of several ways:

- *Direct-verification:* M directly queries the source and receives a reply that $x_i = b$.
- *Comm-verification:* M collects votes from a private ρ-representative committee $\mathcal{C}_i$. M learns that $x_i = b$ if it receives a message saying that $x_i = b$ from at least ρ members of $\mathcal{C}_i$, and a message saying that $x_i = 1 - b$ from fewer than ρ members of $\mathcal{C}_i$.
- *Gossip-verification:* M receives messages from $\beta k + 1$ or more peers, each testifying that it verified $x_i = b$. This suffices since necessarily at least one of these senders must have been an honest peer.

Blacklisting. During an execution, honest peers can *blacklist* Byzantine ones, after identifying a deviation from the behavior expected of an honest peer, and subsequently ignore their messages. A Byzantine peer M' can be blacklisted for several reasons. The most common reason to blacklist is when M' is directly "caught" in a lie about the value of some bit. The two other reasons for blacklisting are as follows.

- **Blacklisting for requesting unnecessary work:** Some of our protocols maintain a known-to-all list of bits. If M' claims that a certain bit x_i is unknown to it and requests to learn it, M can check if x_i is listed at M as *known to all*. If so, M knows that M' must be Byzantine.
- **Blacklisting for over-activity:** Lemma 1 implies that the number of honest peers in our construction of a private ρ-representative committee is bounded from above w.h.p. M' can be blacklisted as Byzantine for being *over-active*, namely, claiming to have been randomly selected to many more committees than expected.

3 Results on the Download Problem

3.1 Deterministic Setting

We first note that Download can be solved trivially by having each peer query all n bits directly from the source. This protocol incurs $\mathcal{Q} = n$, $\mathcal{T} = 1$ and $\mathcal{M} = 0$ and works for $\beta < 1$. However, we can improve the query complexity for $\beta < 1/2$. (Some proofs are deferred to Appendix A.)

▶ **Theorem 2.** *When $\beta < 1/2$, there is a deterministic protocol for Download with $\mathcal{Q} = O(\beta n)$, $\mathcal{T} = \tilde{O}(\beta n)$ and $\mathcal{M} = \tilde{O}(\beta n k^2)$*

The following theorem establishes that one cannot hope to improve the query complexity.

▶ **Theorem 3.** *Any deterministic protocol for the Download problem has $\mathcal{Q} = \Omega(\beta n)$.*

3.2 Randomized setting

Near Query-Optimal Randomized Protocol for $\beta < 1$

We start with a simple randomized algorithm that works for any $\beta < 1$. The problem posed by the randomized model is that the adversary can fail peers online in the randomized setting based on the protocol's progress. This implies that if the protocol appoints some random peer M to query a bit x_i on some round t of the execution but communicate the bit to other peers at a *later* round t', then we cannot rely on the hope that the randomly selected M will be honest, say, with probability $1 - \beta = \varepsilon$, since the adversary gets an opportunity to learn the identity of the chosen M on round t and subsequently corrupt it before round t'. Hence in order for us to benefit from the fact that some peer M is randomly chosen for some sub-task on round t, it is imperative that M completes that sub-task *on the same round.*

The idea used to overcome this difficulty is as follows. Sequentially, for n rounds, do the following. At round i we query bit x_i from the source to a private ρ-representative committee (see Algorithm 1) $\mathcal{C}_i$, i.e., x_i is queried by each (honest) peer in $\mathcal{C}_i$. Then (still on the same round), each peer in $\mathcal{C}_i$ sends the value of x_i to every other peer. Peers not in $\mathcal{C}_i$ might receive incorrect values from the Byzantine peers in $\mathcal{C}_i$. However, if strictly fewer than ρ incorrect values are received, each peer can be confident of the majority as the right answer (w.h.p). In case at least ρ peers sent an incorrect value, or more precisely, in case an honest peer receives at least ρ zeros and at least ρ ones, then peers resort to querying the source for the answer, forcing at least ρ Byzantine peers to reveal themselves as being Byzantine. Choosing ρ optimally results in a query complexity of $O(\frac{n \log n}{\gamma k} + \sqrt{n})$. See Algorithm 2 for the pseudocode.

Algorithm 2 Algorithm `Blacklist_Download` model, Code for peer M.

```
    Output: Array $res$ such that $res[i] = x_i$ for $i = 1, 2, ...n$
 1: $\mathcal{B} \leftarrow \emptyset$                                        ▷ Peers known to be faulty
 2: for $i = 1, 2, \ldots n$ (in separate rounds) do
 3:     Form a private $\rho$-representative committee $\mathcal{C}_i$.        ▷ Parameter $\rho$ is fixed later
 4:     if $M \in \mathcal{C}_i$ then
 5:         $res[i] \leftarrow$ Query($i$), send (vote, $res[i]$) to all peers.
 6:     $S_j \leftarrow$ set of peers not in $\mathcal{B}$ that voted $j$ for $j \in \{0, 1\}$
 7:     if $\min(|S_0|, |S_1|) > \rho$ then
 8:         $res[i] \leftarrow$ Query($i$).
 9:         $\mathcal{B} \leftarrow \mathcal{B} \cup S_{1-res[i]}$
10:     else  $res[i] \leftarrow \arg\max_{j=0,1} |S_j|$.
11: return $res$
```

▶ **Theorem 4.** *When $\beta < 1$, Protocol* `Blacklist_Download` *solves the* Download *problem w.h.p. with $\mathcal{Q} = O\left(\frac{n \log n}{\gamma k} + \sqrt{n}\right)$, $\mathcal{T} = O(n)$ and $\mathcal{M} = O(kn \log n + k^2 \sqrt{\gamma n})$.*

Proof. The correctness follows from the observation that for each bit x_i, each honest peer either (i) heard fewer than ρ votes for one value in $\{0, 1\}$ or (ii) queried x_i. Since $\mathcal{C}_i$ is ρ-representative (w.h.p), the correct bit value would have been reported by at least ρ peers, and we can conclude that an honest peer can verify the correct value of x_i in both cases. Next, We analyze the query complexity of Algorithm 2 and choose ρ optimally.

Queries are made in lines 5 and 8 of Algorithm 2. For peer M, the expected number of queries in line 5 is np where p is the probability of joining a committee. As $n > \gamma k$, we have $np \geq 9 \ln n$. By Chernoff bounds, w.h.p there are no more than $2np$ queries, similar to the proof of Lemma 1. Every time a peer reaches Line 8 and queries the source, the size of its local set $\mathcal{B}$ increases by ρ. Therefore, these queries are performed at most $\frac{\beta k}{\rho}$ times. Therefore, the total number of queries for peer M (w.h.p.) is at most $\mathcal{Q} = \frac{18 n \ln n}{\gamma k} + \frac{8 n \rho}{\gamma k} + \frac{\beta k}{\rho}$, and choosing $\rho = \max\left\{1, k\sqrt{\frac{\gamma \beta}{8n}}\right\}$, we get $\mathcal{Q} = O\left(\frac{n \log n}{\gamma k} + \sqrt{\frac{\beta}{\gamma} \cdot n}\right)$.

By the description of the protocol. the time complexity $\mathcal{T}$ is clearly $O(n)$, the number of iterations. The message complexity $\mathcal{M}$ is calculated as the product of the number of honest peers that join each $\mathcal{C}_i$ (which is $O(\log n + k/\sqrt{n})$) times $O(k)$ (the number of messages sent by each honest peer in $\mathcal{C}_i$) time n (the number of iterations). ◀

Observe that there is a trivial $\Omega(\frac{n}{\gamma k})$ lower bound on $\mathcal{Q}$ (in the case where Byzantine peers crash and do not participate in the execution).

The additional $\sqrt{n}$ term can be neglected whenever $k < \sqrt{n}$ since it is smaller than the lower bound in these cases. Thus, in a wide range of cases, the above protocol is "near-optimal". It is also tolerant against the strongest form of Byzantine adversary, one that even has knowledge of random bits sampled up until the previous round.

Query-Optimal Randomized Protocol for $\beta < 1/3$

The Download protocol of the previous section works when $\beta < 1$ but falls short of yielding optimal query complexity. This section presents a query-optimal protocol for Download when $\beta < 1/3$. For a complete analysis see Appendix A.

The Protocol

Let us first give an overview of the approach. The protocol proceeds in $J_0 = \left\lceil \log_{1/\alpha} \frac{k}{c \log n} \right\rceil$ phases, whose goal is to reduce the number of unknown bits by a *shrinkage factor* $\alpha < 1$. The protocol maintains a number of set variables, updated in each phase, including the following. $\mathcal{K}_M$ (respectively, $\mathcal{U}_M$) is the set of indices i whose value $res[i]$ is already *known* (resp., still *unknown*) to M. At any time during the execution, $\mathcal{K}_M \cup \mathcal{U}_M = \{1, \ldots, n\}$. $res[i] = x_i$ is the Boolean value of x_i for every $i \in \mathcal{K}_M$. (Slightly abusing notation for convenience, we sometimes treat $\mathcal{K}_M$ as a set of *pairs* $(i, res[i])$, i.e., we write $\mathcal{K}_M$ where we actually mean $\mathcal{K}_M \circ res_M$.) Each peer also identifies a set KTA_M of *known-to-all* bits and $\mathcal{I}_M$ of unknown indices for at least one peer. Each phase contains four subroutines, each with a specific goal in mind. First, the `Committee_Work` subroutine forms private committees where each peer M joins committee i if $i \in \mathcal{I}_M$ with some probability. Each member of committee i then reports x_i, and each peer decides whether to accept some or no value (updating $\mathcal{K}_M$ and $\mathcal{U}_M$ accordingly). There is also a blacklisting component in which if a peer belongs to too many committees, it is deemed Byzantine and ignored for the rest of the execution. Second, the `Gossip` subroutine has every peer M report its $\mathcal{K}_M$ to all other peers. If a peer receives at least $\beta k + 1$ reports of the same value for x_i, it accepts it. Third, the second invocation of `Gossip` repeats the reporting of $\mathcal{K}_M$ for every peer M, but this time, in addition to the update of $\mathcal{K}_M$, if a value is reported $2\beta k + 1$ times, it adds it to KTA_M. The motivation behind this second invocation is that if a value is reported $2\beta k + 1$ times, then at least $\beta k + 1$ of those reports are from non-faulty peers. Thus, all peers will accept that value (and add it to $\mathcal{K}_M$). Last, the `Collect_Requests` subroutine is meant to update $\mathcal{I}_M$, i.e., to know which indices are unknown to at least one peer. This subroutine also has a blacklisting component in which if a peer sends a request for index i but $i \in \mathrm{KTA}_M$, M blacklists the requesting peer.

▶ Remark 5. The communication performed in the various steps of the protocol takes more than one time unit in the CONGEST model. Hence, the protocol must also ensure that the different steps are synchronized and that all peers start each step only after the previous step is completed. Relying solely on reports from each peer concerning its progress might lead to deadlocks caused by the Byzantine peers. Hence, the scheduling must be based on the fact that the duration of each step is upper-bounded by the maximum amount of communication the step involves. We omit this aspect from the description of the algorithm.

We next detail the code of the main algorithm and its procedures. (Hereafter, we omit the superscript J when clear from the context.) We denote by *update(i, b)* the function that sets $res[i] = b$, removes i from $\mathcal{U}_m$ and adds it to $\mathcal{K}_M$. We denote by $BlacklistOverWork(w_{max}, M)$ the function that checks the number x of committees M reported to belong to and adds M to $\mathcal{B}$ if $x > w_{max}$. We refer to the first and second invocations of Procedure `Gossip` as `Gossip`(1) and `Gossip`(2) respectively.

Algorithm 3 Algorithm `Gossip_Download`, $\beta < 1/3$, code for peer M.

```
$\mathcal{K}_M \leftarrow \emptyset$                                  ▷ Indices of bits known to $M$
$\text{KTA}_M \leftarrow \emptyset$                               ▷ Indices of bits that are known-to-all
$\mathcal{U}_M \leftarrow \{1, \ldots, n\}$                         ▷ Indices of bits not known to $M$
$\mathcal{I}_M \leftarrow \mathcal{U}_M$                ▷ Indices of bits not known to some non-blacklisted peers
$res \leftarrow \emptyset$                                    ▷ Values of bits known to $M$
$\mathcal{B} \leftarrow \emptyset$                                     ▷ Peers blacklisted by $M$ as Byzantine
$c \leftarrow Z/\gamma$                                  ▷ The parameter $Z$ will be fixed later.
$\alpha \leftarrow \frac{(1+\epsilon)\beta}{(1-\epsilon)(1-2\beta)}$     ▷ shrinkage factor, $\alpha < 1$. The parameter $\epsilon$ will be fixed later.
$J_0 \leftarrow \lceil \log_{1/\alpha} \frac{k}{c \log n} \rceil$                        ▷ Number of phases
for $J = 0, 1, 2, \ldots, J_0 - 1$ (sequentially) do
    Invoke Committee_Work
    Invoke Gossip(1)
    Invoke Gossip(2)
    Invoke Collect_Requests.
for every $i \in \mathcal{U}_M$ do $res[i] \leftarrow$ Query($i$)     ▷ Querying the remaining unknown bits
return $res$
```

Partial Analysis

Sanity checks. Let us start with the two sanity checks needed to ensure the validity of the random selection step and the convergence of the protocol.

▶ **Observation 6.** *For β and ϵ satisfying*

$$\beta < \frac{1-\epsilon}{3-\epsilon} \tag{1}$$

(a) *the chosen shrinkage factor satisfies $\alpha < 1$, and*
(b) *the chosen probability satisfies $p < 1$ for every $0 \le J \le J_0 - 1$.*

Progress tracking variables. Next, we define the notation for the values of the main variables of the protocol during the different phases.

- Denote by $\mathcal{K}_M^J$ (respectively, $\mathcal{U}_M^J$) the value of the set $\mathcal{K}_M$ (resp., $\mathcal{U}_M$) at the beginning of phase J. (Note that it is also the value of $\mathcal{K}_M$ at the end of phase $J-1$)
- Denote by $\mathcal{K}_M^{J,mid}$ (resp., $\mathcal{U}_M^{J,mid}$) the value of the set $\mathcal{K}_M$ (resp., $\mathcal{U}_M$) at the end of the `Gossip`(1) step of phase J.
- Denote by $\mathcal{I}_M^J$ the value of the set $\mathcal{I}_M$ at the beginning of phase J.
- Denote by KTA_M^J the value of the set KTA_M at the end of the `Gossip`(2) step of phase J.

Algorithm 4 Sub routines, code for peer M.

```
procedure Committee_Work
    $\hat{\mathcal{I}}_M \leftarrow \emptyset$                                   ▷ Set of indices whose committees $M$ joins
    $\rho \leftarrow (1-\epsilon)Z\log n/\alpha^J$                               ▷ Also $\rho = (1-\epsilon)p\gamma k$
    $\mathsf{W}_{max} \leftarrow (1+\epsilon)c\log n \cdot \frac{n}{k}$          ▷ Blacklisting "over-active" Byzantine peers
    for every $i = 1, \ldots, n$ sequentially do                                 ▷ Setting up committees
        if $i \in \mathcal{I}_M$ then
            Join the private committee $\mathcal{C}_i$ at random with probability $p = \dfrac{c\log n}{\alpha^J k}$.
            if $M$ was selected to $\mathcal{C}_i$ then
                $\hat{\mathcal{I}}_M \leftarrow \hat{\mathcal{I}}_M \cup \{i\}$.
                if $i \in \mathcal{U}_M$ then
                    $update(i,$ Query$(i))$                                      ▷ Direct-verification
                Send the message (vote ,$i$,$res[i]$) to every other peer.
            Collect votes sent by members of $\mathcal{C}_i$.                    ▷ Ignore messages on bits $i \notin \mathcal{I}_M$.
    for every other peer $M'$ do
        $BlacklistOverWork(\mathsf{W}_{max},\ M')$
    for every $i \in \mathcal{I}_M$ do
        Let $\mathcal{C}_i^M$ be the remaining reduced committee.                ▷ Possibly $\mathcal{C}_i^M \neq \mathcal{C}_i^{M'}$ for $M \neq M'$.
    for every $i \in \mathcal{U}_M$ do                                           ▷ comm-verification
        for $b \in \{0,1\}$ do
            $\psi_b(i) \leftarrow$ number of votes from $\mathcal{C}_i^M$ members for $x_i = b$.
        if $\psi_0(i) \geq \rho$ and $\psi_1(i) < \rho$ then
            $update(i,\ 0)$
        if $\psi_1(i) \geq \rho$ and $\psi_0(i) < \rho$ then
            $update(i,\ 1)$                   ▷ If both $\psi_0(i) \geq \rho$ and $\psi_1(i) \geq \rho$, then$i$ remains unknown

procedure Gossip(GossipNum)
    for every $i \in \mathcal{K}_M$ do
        send the message $(i, res[i])$ to all other peers.
    Receive a list $\mathcal{K}_{M'}$ from every other peer $M'$.
    for every $i \in \mathcal{U}_M$ do
        $\varphi_0(i) \leftarrow |\{M' \mid (i,0) \in \mathcal{K}_{M'}\}|$.
        if $\varphi_0(i) \geq \beta k + 1$ then
            $update(i,\ 0)$
        $\varphi_1(i) \leftarrow |\{M' \mid (i,1) \in \mathcal{K}_{M'}\}|$.
        if $\varphi_1(i) \geq \beta k + 1$ then
            $update(i,\ 1)$
        if GossipNum=2 and ($\varphi_0(i) \geq 2\beta k + 1$ or $\varphi_1(i) \geq 2\beta k + 1$) then
            $\mathrm{KTA}_M \leftarrow \mathrm{KTA}_M \cup \{i\}$

procedure Collect_Requests
    Set $\mathcal{I}_M \leftarrow \mathcal{U}_M$
    Send $\mathcal{U}_M$ to all other peers.
    Collect lists $\mathcal{U}_{M'}$ from all other peers $M'$.
    for every $i = 1, \ldots, n$ do
        $R_U(i) \leftarrow \{M' \mid i \in \mathcal{U}_{M'}\}$.
        if $i \in \mathrm{KTA}_M$ then $\mathcal{B} \leftarrow \mathcal{B} \cup R_U(i)$      ▷ Blacklisting for requesting known-to-all bits
    $\mathcal{I}_M \leftarrow \mathcal{I}_M \cup \bigcup_{M' \notin \mathcal{B}} \mathcal{U}_{M'}$      ▷ Indices to be learned, including $\mathcal{U}_M$ of $M$ itself
```

Note that a bit x_i can be unknown for M and known for M' for two honest peers M and M'. We say that x_i is *unknown* in phase J, and the committee $\mathcal{C}_i$ is *necessary*, if $i \in \mathcal{U}^J_M$ for *some* honest peer M, or equivalently, if $i \in \mathcal{U}^J$, where

$$\mathcal{U}^J = \bigcup_{M \in \mathcal{H}} \mathcal{U}^J_M$$

is the set of indices i for which some honest peers request setting up a committee $\mathcal{C}_i$ and querying in the current phase. A bit x_i is *known* once $i \in \mathcal{K}_M$ for every honest peer M. Also let

$$\mathcal{U}^{J,mid} = \bigcup_{M \in \mathcal{H}} \mathcal{U}^{J,mid}_M \qquad \text{and} \qquad \mathrm{NKTA}^J_M = \{1, \ldots, n\} \setminus \mathrm{KTA}^J_M.$$

Bad events. In an execution ξ of the protocol, there are two types of *bad events*, whose occurrence might fail the protocol. Our analysis is based on bounding the probability of bad events, showing that with high probability, no bad events will occur in the execution, and then proving that in a *clean* execution, where none of the bad events occurred, the protocol succeeds with certainty. The bad events are as follows.

Bad event $\mathcal{EV}_1(J, i)$: In phase J, the committee $\mathcal{C}_i$ selected for an unknown bit x_i is not ρ-representative, for $\rho = \dfrac{(1-\epsilon)Z \log n}{\alpha^J}$, where Z is a parameter of the algorithm that must satisfy some constraints described in Lemmas 7 and 8. (If x_i is already known, then this bad event does not affect the correctness or query complexity of the honest peers, although it might increase the time and message complexity.)

Bad event $\mathcal{EV}_2(J, M)$: In phase J, an honest peer M has $|\hat{\mathcal{I}}^J_M| > \mathsf{W}_{max}$, namely, M joins more than $\mathsf{W}_{max} = \dfrac{(1+\epsilon)c \cdot n \log n}{k}$ committees, and subsequently gets blacklisted.

For an integer $J \geq 0$, call the execution ξ *J-clean* if none of the bad events $\mathcal{EV}_1(j, i)$ or $\mathcal{EV}_2(j, M)$ occurred in it for $0 \leq j \leq J$.

High probability of clean executions. We now argue that with the right choice of parameters ϵ and Z, the probability for the occurrence of any of the bad events is low.

▶ **Lemma 7.** *For any $J \geq 0$, if the execution ξ is $(J-1)$-clean, and the parameters ϵ and Z satisfy*

$$\epsilon^2 Z/2 \geq 2 + \lambda \tag{2}$$

for some constant $\lambda > 0$, then the probability that any of the bad events $\mathcal{EV}_1(J, i)$ occurred in ξ is at most $O(\frac{1}{n^{1+\lambda}})$.

▶ **Lemma 8.** *For any $J \geq 0$, if the execution ξ is $(J-1)$-clean, and the parameters ϵ and Z satisfy*

$$\frac{\epsilon^2}{2+\epsilon} \cdot Z \geq 2 + \lambda \tag{3}$$

for some constant $\lambda > 0$, then the probability that any of the bad events $\mathcal{EV}_2(J, M)$ occurred in ξ is at most $O(\frac{1}{n^{1+\lambda}})$.

The above two lemmas yield the following:

▶ **Corollary 9.** *Consider an execution ξ. If the parameters ϵ and Z satisfy*

$$Z \cdot \min\{\epsilon^2/2\ ,\ \epsilon^2/(2+\epsilon)\}\ \geq\ 2 + \lambda \tag{4}$$

for some constant $\lambda > 0$, then the probability that ξ is clean is at least $1 - O(\frac{\log n}{n^{1+\lambda}})$.

Convergence invariants.

▶ **Lemma 10.** *In a J-clean execution, assuming $\beta < 1/3$, for every honest M,*

$$\mathcal{I}_M^{J+1} \subseteq NKTA_M^J \subseteq \mathcal{U}^{J,mid} \subseteq \mathcal{U}^J \subseteq \mathcal{I}_M^J.$$

We remark that if x_i is known, hence $\mathcal{C}_i$ is not necessary, then the inviting peer is Byzantine, so it may invite only a few honest peers (or none) hence the constructed $\mathcal{C}_i$ is not guaranteed to be ρ-representative, but this will not hurt any honest peer, since, in this case, the honest peers already know x_i and will not listen to the committee.

▶ **Lemma 11.** *In a J-clean execution, for every $J \geq 0$ and every honest M,*
(1) $|\mathcal{U}^{J,mid}| \leq \alpha^{J+1}n$, (2) $|\mathcal{I}_M^J| \leq \alpha^J n$, (3) $|\mathcal{U}^J| \leq \alpha^J n$, (4) $|\mathcal{U}_M^J| \leq \alpha^J n$.

Using these convergence invariants, we get the following theorem.

▶ **Theorem 12.** *When $\beta < 1/3$, Protocol* `Gossip_Download` *solves the* Download *problem w.h.p. with*[4] $\mathcal{Q} = O\left(\frac{n \log^2 n}{\gamma k}\right)$, $\mathcal{T} = O\left(n \log_{\frac{1}{\beta}}\left(\frac{\gamma k}{\log n}\right)\right)$ *and* $\mathcal{M} = O\left(nk^2 \log_{\frac{1}{\beta}}\left(\frac{\gamma k}{\log n}\right)\right)$.

4 Results on the Disjunction Problem

In this section, we consider the problem of computing the Disjunction of the input bits. We first state some basic lower bounds and then present some upper bound results, along with an overview of the building blocks used to design protocols that match the upper bounds. For a complete formal presentation see the full version of the paper.

▶ **Theorem 13.** *When $\beta < 1$, any deterministic protocol for the* Disjunction(δ) *and the Explicit* Disjunction(δ) *problems has* $\mathcal{Q} = \Omega\left(\beta \cdot \boldsymbol{\delta}^{-1} + \frac{(1-\delta)n}{\gamma k}\right)$.

▶ **Theorem 14.** *Any randomized protocol for* Disjunction(δ) *that succeeds with constant probability has* $\mathcal{Q} = \Omega(\frac{1}{\gamma k} \cdot \boldsymbol{\delta}^{-1})$ *in expectation.*

The remainder of this section deals with efficient deterministic protocols for Disjunction and Explicit Disjunction under different settings. A key observation that we rely on is that single round algorithms exhibit similar properties to bipartite expanders. The connection is as follows. One can represent the access pattern of the peers to the input array $\mathcal{X}$ as a bipartite graph $G(L, R, E)$, where L represents the n input bits, R represents the k peers, and an edge $(i, j) \in E$ indicates that M_j queries $\mathcal{X}[i]$. We would like to ensure that if the number of bits set to 1 in $\mathcal{X}$ exceeds some value s, then no matter which set S of indices corresponds to these s 1s, the set $\Gamma(S)$ of neighbors of S in G will contain at least $\beta k + 1$ peers, guaranteeing that *at least one honest peer will query at least one of the set bits of S*. This can be ensured by taking G to be a *Large Set Expander (LSE)*, an expander variant defined formally later on. Not knowing the density δ in advance, we can search for it, starting with the hypothesis that δ is close to 1 (and hence using a sparse LSE and spending a small number of queries), and gradually trying denser LSE's (and spending more queries), until we reach the correct density level allowing some honest peer to discover and expose a set bit. Once the set bit is exposed, we have all the honest peers send the new bit to every other

[4] We remark that our focus was on optimizing query complexity. The $\mathcal{T}$ and $\mathcal{M}$ complexities can be improved further. For example, the current protocol requires the peers to send the entire set of known bits in each iteration, but clearly, it suffices to send the updates.

peer. The peers then query all the bits they received (one per peer) to confirm the answer. The total query complexity per peer is $\tilde{O}(n/k + \boldsymbol{\delta}^{-1} + k)$. Observe that it is near-optimal when $k < \sqrt{n}$. See Theorem 17. For $\beta < 1/2$, we obtain near-optimal query complexity $\tilde{O}(n/k + \boldsymbol{\delta}^{-1})$. The observation leading to this is that one can use expanders as before, and assign vertices to input bits such that for every possible input and every possible set of corrupt peers, strictly more than $k/2$ honest peers query a set bit. Subsequently, whenever more than $1/2$ of the peers found a 1, the remaining honest peers can conclude that the answer is 1, and because of the stronger guarantee, we no longer have to verify all the bits sent by the agents. This algorithm, however, only obtains the Disjunction of the input bits, not the actual index of a set bit. See Theorem 18.

Our definition of LSE ensures that for every possible input configuration of Disjunction with input density δ and every possible set of peers that can be corrupted by Byzantine agents, at least one honest peer reads a set bit. To the best of our knowledge, this exact definition of LSE has not been used in the literature. The definition of samplers [29] to construct asynchronous Byzantine agreement and leader election protocols is the closest to LSE. Roughly speaking, samplers ensure there are at most δ fraction of the input bits x such that their neighborhood has β fraction of Byzantine nodes, for every possible choice of corruptions that the adversary can make. Even though our definitions are different, we use similar techniques (the probabilistic method) to show their existence.

▶ **Definition 15** (Large Set Expander (LSE)). A bipartite graph $G(L, R)$ is an (n, k, β, δ)-*Large Set Expander* (or (n, k, β, δ)-LSE) if $n = |L|, k = |R|$ and $|\Gamma(S)| > \beta k$ for all $S \subseteq L$ with $|S| \geq n\delta$.

Informally, a large set expander is such that for every large enough subset S, i.e., $S \subseteq L$ and $|S| \geq \delta n$, its neighborhood cannot be covered fully by any subset of βk vertices, i.e., $|\Gamma(S)| > \beta k$. The definition of an LSE is similar to that of expander graphs and we use a similar probabilistic analysis to prove their existence. We formalize this in the lemma below.

▶ **Lemma 16.** *There exists a bipartite graph $G(L, R)$ that is a (n, k, β, δ) Large Set Expander such that, (1) Every vertex in L has degree at most d, and (2) Every vertex in R has degree at most $\frac{2nd}{k}$, for all d satisfying* $d > \max\left\{\frac{1+\log(e \cdot \boldsymbol{\delta}^{-1})}{\log\frac{1}{\beta}} + \frac{\beta k}{\delta n} \cdot \frac{\log\frac{e}{\beta}}{\log\frac{1}{\beta}}, \frac{3k\ln 2k}{n}\right\}$.

We use the existence of large set expanders to design algorithms that achieves the results stated in Theorems 17 and 18.

▶ **Theorem 17.** *When $\beta < 1$, There exists a protocol that solves* Disjunction *with* $\mathcal{Q} = O\left(\frac{n}{k} \cdot \left(\log_{\frac{1}{\beta}}(e^2\boldsymbol{\delta}^{-1}) + \log k\right) \cdot \log \boldsymbol{\delta}^{-1} + \boldsymbol{\delta}^{-1} \cdot (\beta \log_{\frac{1}{\beta}} \frac{e}{\beta}) + k\right)$, $\mathcal{T} = O(\log n)$ *and* $\mathcal{M} = O(\beta k^2 \log n)$.

Ignoring log factors and constants dependent on β, the resulting query complexity is $\mathcal{Q} = \tilde{O}(n/k + \boldsymbol{\delta}^{-1} + k)$, essentially matching the lower bound (except for the additive k term). The constant factors increase as β gets closer to 1 and reduce to the naive algorithm when $\beta = 1 - 1/k$. We improve on that in the following result, albeit with the cost of β being at most $1/2$.

▶ **Theorem 18.** *When $\beta < 1/2$, There exists a protocol that solves* Disjunction *with* $\mathcal{Q} = O\left(\frac{n \log n}{k \log(2/(2\beta+1))} + \frac{1}{\log(2/(2\beta+1))} \cdot \boldsymbol{\delta}^{-1} + \log^2 n\right)$, $\mathcal{T} = O(\log n)$ *and* $\mathcal{M} = O(\beta k^2 \log n)$.

Allowing randomization in the protocol design, we achieve the following result.

▶ **Theorem 19.** *When $\beta < 1$, There exists a protocol that w.h.p. solves* Disjunction *with* $\mathcal{Q} = O\left(\frac{\log n}{\gamma k} \cdot \boldsymbol{\delta}^{-1} + \frac{\log k \log n \log(1/\boldsymbol{\delta})}{\gamma}\right)$, $\mathcal{T} = O\left(\log k \cdot \log \boldsymbol{\delta}^{-1}\right)$ *and* $\mathcal{M} = O(k^2 \log k \cdot \log \boldsymbol{\delta}^{-1})$.

5 Related Work

To the best of our knowledge, we are the first to study retrieval problems in the DR model, as defined above. We now provide a description of related studies.

As discussed earlier, Byzantine resilience research was largely limited to a few problems like Byzantine Agreement, Byzantine Broadcast, State Machine Replication, etc. More recently, we have seen many investigations of Byzantine resilience in other problems and models. Quite naturally, it has been explored in P2P settings to ensure robust membership sampling [9] and resilient P2P overlay design [21, 3]. Apart from that, Byzantine resilience was explored in the context of mobile agents [17, 10, 15] and graph algorithms [5]. In the last decade, there was quite a bit of interest in Byzantine resilient learning, starting with multi-armed bandit problems [6]. Finally, there was a recent flurry of works inspired by the popularity of Byzantine resilient optimization algorithms in federated and distributed learning [41, 8, 25, 19, 42, 20].

Byzantine Reliable Broadcast (BRB) was first introduced by Bracha [11]. In BRB, a designated sender holds a message M, and the goal is for every honest peer to output the same M' that must uphold $M' = M$ if the sender is honest. The Download problem can be viewed as a variant of BRB, where the sender is always honest but has no computational powers and is passive (read-only), and peers are always required to output the correct message M. These differences make solving Download different than solving BRB. One easy-to-see difference in results is that Download can be solved trivially even when $1/3 \leq \beta < 1$ and there are no authenticated messages, whereas BRB can not be solved under the same conditions [18]. Another difference is that state-of-the-art BRB protocols like [2] where the sender uses error-correcting codes and collision-resistant hash functions are inapplicable (when considering the source to be the sender). In optimal *balanced* BRB protocols like in [2], the sender sends $O(\frac{n}{k})$ bits to each peer whereas Theorem 3 shows that Download requires $\Omega(\beta n)$ queries (the difference stems from the inability of the source to perform computations).

Most works on Byzantine resilience have focused on models and problems where the data is integrated into the network, making it difficult to get Byzantine resilience past $\beta < 1/3$ or $\beta < 1/2$. However, there have been some exceptions that were observed quite early in the Byzantine resilience literature, like authenticated broadcast [18] that can be achieved for any $\beta < 1$. More recently, the power of decoupling data and computing came into play in the context of mobile agents. The *gathering* problem [17], where mobile agents must gather at one location, can be solved for all fixed $\beta < 1$. Crucially, the honest agents can explore every part of the graph. The Byzantine agents do not control any portion of the graph.

Our work can be viewed as a step towards understanding the power of oracles with the data source playing that role. The use of oracles (also called probes, queries, etc.) has been widespread in classical computing with references dating back to the early seventies [40, 39, 32, 31, 26]. See [28] for an excellent treatment of the various structural complexity theory results that have been obtained through oracles. The power of oracles has been explored in distributed computing as well in the context of overcoming challenges posed by failures in asynchronous settings [35]. On the broader algorithmic front, the property testing model [24] can be viewed as using oracles to access data that is only available through expensive queries.

In essence, we have proposed a hybrid combination of two communication technologies – querying the source and P2P message passing. Such hybrid combinations leading to overall improvements is not new [27]. Friedman et al. [23] studied distributed computing aided by an external entity that they called cloud. They studied asynchronous consensus with the cloud providing a common compare-and-swap (CAS) register access. More recently, Afek et al. [1] introduced the computing with cloud (CWC) model wherein traditional distributed computing models were augmented with one or more cloud nodes that are typically connected to several regular nodes.

The notion of an External Data Source that multiple peers can access is reminiscent of the PRAM model [22, 30] where all processors could access a shared memory. Unfortunately, there has been no work on Byzantine resilience in the PRAM setting. This is not surprising because the PRAM setting allows writing over the shared memory, and Byzantine processors can easily overwrite portions of the input, thereby making it impossible to solve problems in the exact sense.

6 Directions for future work

Our framework adds Byzantine resilience to standard distributed computing with the help of an External Data Source, an entity external to the network. We initiated this study through deterministic and randomized models, focusing on the Download and Disjunction problems, and developing several algorithms, tools, and techniques. Our emphasis was on optimizing the query complexity but also considered time and message complexities. Extending our work to other model variations and/or broader classes of problems like graph and geometric problems, data analytics and peer learning problems are natural next steps.

Our work has shown that this framework is well-suited for Byzantine resilience owing to decoupling of data and computation that lends well to "trust, but verify" techniques in an algorithmically rigorous manner. It will be interesting to see the limits to which Byzantine resilience can be pushed in this framework.

This framework can be interpreted in multiple ways and applied to a wide variety of contexts. Ideas from oracle based computation such as property testing [24] can be easily adapted to our context. One can also envision variants in which the External Data Source offers a richer set of services that may include computation or data re-organization at its end that the peers may need to pay for. Such dynamics can potentially uncover many algorithmic and game theoretic issues like pricing mechanisms and coalition formation. Our approach is thus relevant in contexts like blockchain oracles [7, 12] where a distributed set of peers wish to perform computation on multiple public data sources at different locations (like news outlets, government portals, think-tank reports, etc.) with disparate access costs, access controls and varying levels of trustworthiness. We therefore believe that our work will lead to several other follow-up work exploring all these variations.

In this paper we studied a strong adversarial model. If the source is allowed to provide also a source of global randomness, then our results may be improved further. Specifically, with such service, one can deploy committees guaranteed to have an honest majority w.h.p., which may lead to efficient algorithms for additional problems.

References

1 Yehuda Afek, Gal Giladi, and Boaz Patt-Shamir. Distributed computing with the cloud. In *23rd Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Springer, 2021. doi:10.1007/978-3-030-91081-5_1.

2 Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Balanced byzantine reliable broadcast with near-optimal communication and improved computation. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, pages 399–417, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519270.3538475.

3 John Augustine, Soumyottam Chatterjee, and Gopal Pandurangan. A fully-distributed scalable peer-to-peer protocol for byzantine-resilient distributed hash tables. In *34th ACM SPAA*, pages 87–98, 2022. doi:10.1145/3490148.3538588.

4 John Augustine, Anisur Rahaman Molla, and Gopal Pandurangan. Byzantine agreement and leader election: From classical to the modern. In *ACM PODC*, pages 569–571, 2021. doi:10.1145/3465084.3467484.

5 John Augustine, Anisur Rahaman Molla, Gopal Pandurangan, and Yadu Vasudev. Byzantine Connectivity Testing in the Congested Clique. In *36th DISC*, pages 7:1–7:21, 2022. doi:10.4230/LIPICS.DISC.2022.7.

6 Baruch Awerbuch and Robert Kleinberg. Competitive collaborative learning. *JCSS*, 74(8):1271–1288, 2008. doi:10.1016/J.JCSS.2007.08.004.

7 Abdeljalil Beniiche. A study of blockchain oracles, 2020. arXiv:2004.07140.

8 Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. In *NeurIPS*, pages 119–129, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/f4b9ec30ad9f68f89b29639786cb62ef-Abstract.html.

9 Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: Byzantine resilient random membership sampling. *Computer Networks*, 53(13):2340–2359, 2009. doi:10.1016/J.COMNET.2009.03.008.

10 Sébastien Bouchard, Yoann Dieudonné, and Anissa Lamani. Byzantine gathering in polynomial time. *Distributed Comput.*, 35(3):235–263, 2022. doi:10.1007/S00446-022-00419-9.

11 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information & Computation*, 75:130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

12 Giulio Caldarelli. Overview of blockchain oracle research. *Future Internet*, 14:175, June 2022. doi:10.3390/fi14060175.

13 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *3rd Symp. on Operating Systems Design and Implementation*, OSDI, pages 173–186. USENIX Assoc., 1999. URL: https://dl.acm.org/citation.cfm?id=296824.

14 Soma Chaudhuri, Maurice Erlihy, Nancy A Lynch, and Mark R Tuttle. Tight bounds for k-set agreement. *J. ACM*, 47(5):912–943, 2000. doi:10.1145/355483.355489.

15 Arnhav Datar, Nischith Shadagopan M. N, and John Augustine. Gathering of anonymous agents. In *AAMAS*, pages 1457–1465, 2023. doi:10.5555/3545946.3598798.

16 Arnhav Datar, Arun Rajkumar, and John Augustine. Byzantine spectral ranking. In *NeurIPS*, volume 35, pages 27745–27756, 2022.

17 Yoann Dieudonné, Andrzej Pelc, and David Peleg. Gathering despite mischief. *ACM Trans. Algorithms*, 11(1), August 2014. doi:10.1145/2629656.

18 D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Computing*, 12(4):656–666, 1983. doi:10.1137/0212045.

19 El-Mahdi El-Mhamdi, Sadegh Farhadkhani, Rachid Guerraoui, Arsany Guirguis, Lê-Nguyên Hoang, and Sébastien Rouault. Collaborative learning in the jungle (decentralized, byzantine, heterogeneous, asynchronous and nonconvex learning). In *NeurIPS*, pages 25044–25057, 2021. URL: https://proceedings.neurips.cc/paper/2021/hash/d2cd33e9c0236a8c2d8bd3fa91ad3acf-Abstract.html.

20 Sadegh Farhadkhani, Rachid Guerraoui, Lê Nguyên Hoang, and Oscar Villemaud. An equivalence between data poisoning and byzantine gradient attacks. In *ICML*, pages 6284–6323. PMLR, 2022. URL: https://proceedings.mlr.press/v162/farhadkhani22b.html.

21 Amos Fiat, Jared Saia, and Maxwell Young. Making chord robust to byzantine attacks. In *13th ESA*, pages 803–814. Springer, 2005. doi:10.1007/11561071_71.

22 Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proc. Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114–118, 1978. doi:10.1145/800133.804339.

23 Roy Friedman, Gabriel Kliot, and Alex Kogan. Hybrid distributed consensus. In *Proc. 17th Int. Conference on Principles of Distributed Systems, OPODIS 2013.*, pages 145–159, 2013. doi:10.1007/978-3-319-03850-6_11.

24 Oded Goldreich. *Introduction to Property Testing.* Cambridge Univ. Press, 2017.

25 Nirupam Gupta and Nitin H. Vaidya. Fault-tolerance in distributed optimization: The case of redundancy. In *ACM PODC*, pages 365–374, 2020. doi:10.1145/3382734.3405748.

26 Péter Hajnal. An ω (n 4/3) lower bound on the randomized complexity of graph properties. *Combinatorica*, 11:131–143, 1991.

27 Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting data center networks with multi-gigabit wireless links. *SIGCOMM Comput. Commun. Rev.*, 41(4):38–49, August 2011. doi:10.1145/2018436.2018442.

28 Lane A Hemaspaandra and Mitsunori Ogihara. The complexity theory companion. *Acm Sigact News*, 32(4):66–68, 2001. doi:10.1145/568425.568436.

29 Bruce M. Kapron, David Kempe, Valerie King, Jared Saia, and Vishal Sanwalani. Fast asynchronous byzantine agreement and leader election with full information. In *Proc. 19th ACM-SIAM Symp. on Discrete Algorithms, SODA 2008*, pages 1038–1047, 2008. URL: http://dl.acm.org/citation.cfm?id=1347082.1347196.

30 Richard M Karp. *A survey of parallel algorithms for shared-memory machines.* University of California at Berkeley, 1988.

31 Valerie King. Lower bounds on the complexity of graph properties. In *Proc. 20th ACM Symposium on Theory of Computing*, STOC '88, pages 468–476, 1988. doi:10.1145/62212.62258.

32 Daniel J Kleitman and David Joseph Kwiatkowski. Further results on the aanderaa-rosenberg conjecture. *Journal of Combinatorial Theory, Series B*, 28(1):85–95, 1980. doi:10.1016/0095-8956(80)90057-X.

33 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982. doi:10.1145/357172.357176.

34 Silvio Micali and Tal Rabin. Collective coin tossing without assumptions nor broadcasting. In *CRYPTO*, pages 253–266, Berlin, Heidelberg, 1991. Springer. doi:10.1007/3-540-38424-3_18.

35 Achour Mostefaoui, Eric Mourgaya, and Michel Raynal. An introduction to oracles for asynchronous distributed systems. *Future Generation Computer Systems*, 18(6):757–767, 2002. doi:10.1016/S0167-739X(02)00048-1.

36 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, April 1980. doi:10.1145/322186.322188.

37 Michael O. Rabin. Randomized byzantine generals. In *24th FOCS*, pages 403–409, 1983. doi:10.1109/SFCS.1983.48.

38 R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978. doi:10.1145/359340.359342.

39 Ronald L. Rivest and Jean Vuillemin. On recognizing graph properties from adjacency matrices. *Theoretical Computer Science*, 3(3):371–384, 1976. doi:10.1016/0304-3975(76)90053-0.

40 Arnold L. Rosenberg. On the time required to recognize properties of graphs: a problem. *SIGACT News*, 5(4):15–16, October 1973. doi:10.1145/1008299.1008302.

41 Lili Su and Nitin H. Vaidya. Multi-agent optimization in the presence of byzantine adversaries: Fundamental limits. In *American Control Conf, ACC*, pages 7183–7188. IEEE, 2016. doi:10.1109/ACC.2016.7526806.

42 Lili Su and Nitin H. Vaidya. Byzantine-resilient multiagent optimization. *IEEE Trans. Autom. Control.*, 66(5):2227–2233, 2021. doi:10.1109/TAC.2020.3008139.

A Some missing proofs

Proof of Theorem 2

Proof. To perform Download deterministically, we use *public majority committees*. In this type of committee, *public* means that every peer knows the committee members, and *majority* that the committee is guaranteed to have a strict majority of honest members. The algorithm creates n public majority committees (one per input bit). The committee $\mathcal{C}_i$ is constructed by assigning it $2\beta k + 1$ for $1 \leq i \leq n$ in a round robin fashion with wrap-around. See Algorithm 5. This ensures that

1. each committee gets $2\beta k + 1$ members, thereby establishing majority, and
2. each peer appears in at most $O(\beta n + n/k)$= $O(\beta n)$ committees (since $\beta \geq 1/k$).

Algorithm 5 Elect Public Majority Committee $\mathcal{C}_i$.

```
for $0 \leq j < 2\beta k + 1$ do
    Assign peer $(i-1)(2\beta k + 1) + j \pmod{k} + 1$, to $\mathcal{C}_i$.
```

The key observation is that it suffices if each bit i is queried by a public majority committee C_i since when such a committee sends votes on the value bit to every other peer, each other (honest) peer can trust the majority vote of the committee. Constructing public majority committees is done as described in Algorithm 5, and complexity measures follow from the properties of the construction (see Sect. 2). ◀

Proof of Theorem 3

Proof. To establish this, we prove a slightly stronger claim. Consider a deterministic protocol $\mathcal{P}$ for the Download problem. For an n-bit input $\mathcal{X}$, let $\mathcal{E}(\mathcal{X})$ denote the (unique) execution of $\mathcal{P}$ on $\mathcal{X}$ in which none of the peers has failed. Then, the following holds.

▶ **Lemma 20.** *For every $\mathcal{X}$, every bit x_i ($1 \leq i \leq n$) is queried by at least $\beta k + 1$ peers during the execution $\mathcal{E}(\mathcal{X})$.*

Proof. Towards contradiction, suppose there exists an input $\mathcal{X} = \{x_1, \ldots, x_n\}$ and an index $1 \leq i \leq n$ such that in the execution $\mathcal{E} = \mathcal{E}(\mathcal{X})$, the set $\hat{M}$ of peers that queried the bit x_i is of size $|\hat{M}| \leq \beta k$. Without loss of generality, let $x_i = 0$.

The adversary can now apply the following strategy. It first simulates the protocol $\mathcal{P}$ on $\mathcal{X}$ and identifies the set $\hat{M}$. It now generates an execution $\mathcal{E}'$ similar to $\mathcal{E}$ except for the following changes: (a) The input $\mathcal{X}' = \{x'_1, \ldots, x'_n\}$ in $\mathcal{E}'$ is the same as $\mathcal{X}$ except that $x'_i = 1$. (b) The peers of $\hat{M}$ are Byzantine; all other peers are honest. (c) Each Byzantine peer $M \in \hat{M}$ behaves according to $\mathcal{P}$ except that it pretends that $x'_i = 0$, or in other words, it behaves as if the input is $\mathcal{X}$ (and the execution is $\mathcal{E}$).

One can verify (e.g., by induction on the rounds) that the honest peers cannot distinguish between the executions $\mathcal{E}$ and $\mathcal{E}'$. Therefore, they end up with the same output in both executions. This contradicts the fact that their output in $\mathcal{E}$ must be $\mathcal{X}$, and their output in $\mathcal{E}'$ must be $\mathcal{X}'$. ◀

The lemma implies that for every input $\mathcal{X}$, the total query complexity of the protocol is greater than βkn. Theorem 3 follows. ◀

In the remainder of this section we present the analysis of the main body of Theorem 12.

When M joins (in Procedure `Committee_Work`) the committee $\mathcal{C}_i$ for some $i \in \mathcal{U}^J_M$, M is required to *actively* query the source for the value of x_i. We then say that $\mathcal{C}_i$ is an *active committee* for M. (In contrast, when M joins a committee $\mathcal{C}_i$ for $i \in \mathcal{K}^J_M$, it costs it nothing since it already has the value of x_i stored in $res[i]$, so it does not need to spend another query.) We define the following size variables.

- Let $\tilde{n}^J_M$ denote the number of *active* committees for M in phase J.
- Let $\hat{n}^J_M = |\hat{\mathcal{I}}^J_M|$ denote the total number of committees that M joins by Procedure `Committee_Work` in phase J. (Note that $\tilde{n}^J_M \le \hat{n}^J_M$)
- Let $n^J_M = |\mathcal{I}^J_M|$ denote the total number of requests received by M by Procedure `Collect_Requests` in phase J.

▶ **Lemma 21.** *If some honest M adds i to its set KTA^J_M of known-to-all bits at the end of the* `Gossip`(2) *step of phase J, then $i \in \mathcal{K}^{J+1}_{M'}$ for every honest M'.*

▶ Note 22. the sets KTA_M might not be all equal. Namely, every honest peer might be aware of a different subset of the known-to-all bits. Note, however, that as shown later in Lemma 27, the sets KTA_M of all honest peers contain the set CORE discussed in the high-level overview, and the fast growth of CORE is essentially the cause for the fast shrinkage of the set of unknown bits.

Proof. Suppose $i \in \mathrm{KTA}^J_M$ for some honest M. Then in `Gossip`(2) of phase J, M counted at least $2\beta k + 1$ messages containing (i, b) (for $b \in \{0, 1\}$). At least $\beta k + 1$ of these messages were sent by honest peers, and therefore, in the `Gossip`(2) step of phase J, all honest peers will count at least $\beta k + 1$ messages containing (i, b). Consequently, every honest peer M' will move i to $\mathcal{K}_{M'}$ at that step, so $i \in \mathcal{K}^{J+1}_{M'}$. ◀

Properties of clean executions.

▶ **Observation 23.** *In a J-clean execution, if $i \in \mathcal{U}^J$ (i.e., x_i is still unknown in phase J), then for every honest peer M, the reduced committee $\mathcal{C}^M_i$ is ρ-representative.*

▶ Remark 24. Note that once a committee is selected, the adversary can corrupt all of its members in the very next round. By then, however, the committee had completed its querying and communication actions, so the fact that it is no longer representative does not harm the execution. Note also that the need to complete all committee actions in a single round is the reason why it is required to perform the querying sequentially, spending a round for each bit x_i. The querying operations of all committees could, in principle, be parallelized, but the subsequent communication step might require more than a single round in the CONGEST model, giving the adversary an opportunity to intervene and corrupt an entire committee before it has completed sending its messages.

Note that those bits that were not moved from $\mathcal{U}_M$ to $\mathcal{K}_M$ during the main phases J of the protocol were directly-verified in the final step of the protocol. This implies the following.

▶ **Observation 25.** *By the end of the execution, every honest peer has the value $res[i]$ for every bit x_i.*

It remains to show that for every x_i, the $res[i]$ value obtained by each honest peer is correct.

▶ **Lemma 26.** *In a J-clean execution, whenever an honest peer learns an input bit x_i in phases 0 to J, the learned value $res[i]$ is correct.*

Proof. Consider an input bit x_i. Order the honest peers that learned x_i during phases 0 to J according to the time by which they acquired x_i. the proof is by induction on this order.

For the induction basis, note that the first peer to acquire x_i must have directly verified it, so the value it has obtained is clearly correct.

Now consider the t-th peer M in this order, and suppose M knows that $x_i = b$. there are several cases to consider.

Case 1. M directly-verified x_i, either on the last step of the protocol or in Procedure `Committee_Work` during some phase J. Then again, $res[i]$ is clearly correct.

Case 2. M comm-verifies x_i, in Procedure `Committee_Work`. Then M found $\psi_b(i) \geq \rho$ and $\psi_{1-b}(i) < \rho$. Since the execution is J-clean, $\mathcal{C}_i^M$ is ρ-representative by Lemma 23. This implies that if $x_i = 1 - b$ then all the honest peers in $\mathcal{C}_i^M$ would return $1 - b$, and M would find $\psi_{1-b}(i) \geq \rho$, which did not happen. Hence, $x_i = b$.

Case 3. M gossip-verifies x_i, in the `Gossip`(1) or `Gossip`(2) step. Then M has received messages from $\beta k + 1$ or more peers stating that they already know that $x_i = b$. At least one of those peers, M', is honest, and it acquired x_i prior to M. Hence the inductive hypothesis applies to it, yielding that indeed $x_i = b$. ◀

Proofs of Convergence invariants

Proof of Lemma 10.

Proof. Consider a bit index $i \notin \mathrm{NKTA}_M^J$. Then x_i is marked known-to-all by M in the `Gossip`(2) step. Consequently, M ignores x_i in phase J even if it receives it in some request message in Procedure `Collect_Requests`. Hence $i \notin \mathcal{I}_M^{J+1}$. The first containment follows

Consider a bit index $i \in \mathrm{NKTA}_M^J$. Then x_i is not listed as known-to-all in M, i.e., $i \notin \mathrm{KTA}_M$, so M had $\varphi_0(i) \leq 2\beta k$ and $\varphi_1(i) \leq 2\beta k$.

Let $b = x_i$, i.e., the correct value of x_i, and let $0 \leq \delta \leq 1$ be the fraction of faulty peers that reported knowing i. Since the execution is J-clean, by Lemma 26, we know that $\varphi_{1-b}(i) \leq \delta\beta k$. Therefore $\varphi_0(i) + \varphi_1(i) \leq (2 + \delta)\beta k$. Hence, the number of peers that informed M that they do not know x_i satisfies $k-(\varphi_0(i)+\varphi_1(i)) \geq (1-(2+\delta)\beta)k > (1-\delta)\beta k$, where the second inequality follows since $\beta < 1/3$.

Hence, there is at least one honest peer M' that did not send x_i as part of its $\mathcal{K}_{M'}^{J,mid}$, so $i \in \mathcal{U}_{M'}^{J,mid}$, and hence $i \in \mathcal{U}^{J,mid}$. The second containment follows.

The next containment follows from the fact that for an honest peer M, $\mathcal{U}_M$ is monotone, decreasing in time.

Consider an index $i \in \mathcal{U}^J$. Then some honest $M' \in \mathcal{H}$ has $i \in \mathcal{U}_{M'}^J$. This has two implications when $J \geq 1$. First, M' will send a request to learn i in Procedure `Collect_Requests` of phase $J-1$. Second, by Lemma 21 $i \notin \mathrm{KTA}_M^{J-1}$(otherwise $i \in \mathcal{K}_{M'}^J$). Hence M will respect the request by M' and add i to I_M^J. When $J = 0$, $\mathcal{U}^J = \{1, \ldots, n\} = \mathcal{U}_M^J = \mathcal{I}_M^J$. The fourth containment follows. ◀

Define the *core of 2-common-knowledge* after phase J as follows. For every index i, let $num_V^J(i)$ denote the number of honest peers M that comm-verified i and updated it in Procedure `Committee_Work` of phase J. Then

$$\mathsf{CORE}^J \;=\; \{i \mid num_V^J(i) \geq \beta k + 1\}.$$

The name is justified by the following lemma.

▶ **Lemma 27.** *If $i \in \mathsf{CORE}^J$ then, $i \in \mathcal{K}_M^{J,mid}$ and $i \in KTA_M^J$, for every honest peer M*

Proof. Consider an index $i \in \mathsf{CORE}^J$. By definition, x_i was comm-verified by at least $\beta k + 1$ honest peers during Procedure `Committee_Work` of phase J. Each of these peers will send i (along with its value) to every other peer during the `Gossip`(1) step. Subsequently, at the end of this round, $i \in \mathcal{K}_M^{J,mid}$ for every honest M. Consequently, in `Gossip`(2) of phase J, *all* honest peers will report knowing x_i , so every honest peer M will add it to KTA_M^J ◀

Proof of Lemma 11.

Proof. We first prove part (1), by considering iteration $J \geq 0$ and bounding $|\mathcal{U}^{J,end}|$ at its end.

The purpose of blacklisting Byzantine peers that claim to participate in too many committees, via defining reduced committees, is to curb the influence of the Byzantine peers on votes, by bounding the extent of *Byzantine infiltration* into committees. For every honest peer M and Byzantine peer M', denote by $\mathsf{BI}_M(M')$ the number of reduced committees $\mathcal{C}_i^M$ that M' claimed to belong to. (Note that for peers M' that were not blacklisted, this value is the same as $\mathsf{Work}(M')$.) Denote the total number of Byzantine infiltrations into reduced committees of M by $\mathsf{BI}_M = \sum_{M' \in \mathcal{B}} \mathsf{BI}_M(M')$. Denote the total number of Byzantine infiltrations into reduced committees of honest peers by $\mathsf{BI} = \sum_{M \in \mathcal{H}} \mathsf{BI}_M$. By the way M constructs the reduced committees in Procedure `Committee_Work`, every peer appears in at most W_{max} reduced committees of M, hence $\mathsf{BI}_M \leq \beta k \cdot \mathsf{W}_{max}$, and therefore

$$\mathsf{BI} \;\leq\; \gamma k \cdot \mathsf{BI}_M \;\leq\; \gamma k \cdot \beta k \cdot \mathsf{W}_{max} \;=\; (1+\epsilon)c\beta\gamma \cdot kn\log n.$$

Consider a bit $x_i \in \mathcal{U}^J$. By the fourth containment of Lemma 10, $x_i \in \mathcal{I}_M^J$ for every honest peer M. Hence every honest M will set up a committee $\mathcal{C}_i$, which will be ρ-representative since the execution is J-clean.

A necessary condition for x_i to remain in $\mathcal{U}^{J,mid}$ is that at most βk honest peers directly verify it in Procedure `Committee_Work` of phase J. This is because otherwise,$i \in \mathsf{CORE}^J$ and by lemma 27, it will belong to $\mathcal{K}_M^{J,mid}$ for every honest M.

Hence, in order to keep i in $\mathcal{U}^{J,mid}$, the adversary must prevent at least $(1-2\beta)k$ honest peers from directly- or comm-verifying x_i. To achieve that, at least ρ Byzantine peers must infiltrate the reduced committee $\mathcal{C}_i^M$ for at least $(1-2\beta)k$ honest peers. This incurs at least $(1-2\beta)k\rho$ work. Hence, the number of bits x_i for which this can happen is at most

$$|\mathcal{U}^{J,mid}| \;\leq\; \frac{\mathsf{BI}}{(1-2\beta)k\rho} \;\leq\; \frac{(1+\epsilon)c\beta\gamma \cdot kn\log n}{(1-2\beta)k \cdot (1-\epsilon)Z\log n/\alpha^J} \;=\; \frac{(1+\epsilon)\beta Z \cdot \alpha^J n}{(1-2\beta)(1-\epsilon)Z} \;=\; \alpha{\cdot}\alpha^J n\,,$$

where the last equality is by the definition of α. This yields Part (1).

By Lemma 10, Part (2) follows from part (1) upon noting that $n_M^J = |\mathcal{I}_M^J| \leq |\mathcal{U}^{J-1,mid}|$, and Part (3) follows from part (2). Part (4) follows from part (3), noting that $\mathcal{U}_M^J \subseteq \mathcal{U}^J$. ◀

Proofs of high probability of clean executions

Proof of Lemma 7.

Proof. We first show that for every bit x_i, $\mathbb{P}[\mathcal{EV}_1(J,i)] \leq 1/n^{2+\lambda}$.

Consider an index $i \in \mathcal{U}^J$. By Lemma 10, $\mathcal{U}^J \subseteq \mathcal{I}_M^J$, and hence $i \in \mathcal{I}_M^J$,

Therefore, all honest peers join the committee $\mathcal{C}_i$ with probability p. Hence, denoting the number of honest peers in $\mathcal{C}_i$ by X,

$$\mathbb{E}[X] \;=\; p|\mathcal{H}| \;\geq\; p \cdot \gamma k \;=\; \frac{\gamma c\log n}{\alpha^J} = \frac{Z\log n}{\alpha^J}.$$

$$\mathbb{P}[\mathcal{EV}_1(J,i)] \;=\; \mathbb{P}[X<\rho] \;=\; \mathbb{P}[X \le (1-\epsilon)\cdot Z\log n/\alpha^J] \le \mathbb{P}[X \le (1-\epsilon)\mathbb{E}[X]] \tag{5}$$

By Chernoff's bound,

$$\mathbb{P}[X \le (1-\epsilon)\mathbb{E}[X]] \le \exp\left(-\frac{\epsilon^2\mathbb{E}[X]}{2}\right) \le \exp\left(-\frac{\epsilon^2}{2}\cdot\frac{Z\log n}{\alpha^J}\right), \tag{6}$$

and by Eq. (2) it follows that

$$\mathbb{P}[\mathcal{EV}_1(J,i)] \;\le\; \exp\left(-(2+\lambda)\cdot\frac{\log n}{\alpha^J}\right) \;\le\; n^{-2-\lambda}.$$

By the union bound, the probability that *any* bad event of type $\mathcal{EV}_1$ occurred in the execution is at most $O(\frac{1}{n^{1+\lambda}})$. ◀

Proof of Lemma 8.

Proof. We first show that for every honest peer M, $\mathbb{P}[\mathcal{EV}_2(J,M)] \le 1/n^{8/3}$. The bad event $\mathcal{EV}_2(J,M)$ occurs if $\hat{n}^J_M > \mathsf{W}_{max}$ in phase J. In Procedure `Committee_Work`, M tries (randomly) to join the committee $\mathcal{C}_i$ for every $x_i \in \mathcal{I}^J_M$, hence $\mathbb{E}[\hat{n}^J_M] = pn^J_M$. Applying Lemma 11(2), we get that

$$\mathbb{E}[\hat{n}^J_M] \;\le\; p\alpha^J n$$

We introduce a variable $X \in (0,1]$ such that

$$\mathbb{E}[\hat{n}^J_M] \;=\; X\cdot p\alpha^J n \;=\; X\cdot c\log n\cdot\frac{n}{k} \;=\; \frac{X\cdot \mathsf{W}_{max}}{1+\epsilon}.$$

We can see now that

$$\mathbb{P}[\mathcal{EV}_2(J,M)] \;=\; \mathbb{P}[\hat{n}^J_M > \mathsf{W}_{max}] \;\le\; \mathbb{P}\left[\hat{n}^J_M > \frac{1+\epsilon}{X}\cdot\mathbb{E}[\hat{n}^J_M]\right]$$

Using the variation of Chernoff's bound that says that, for $\delta > 0$,

$$\mathbb{P}\left[A > (1+\delta)\mathbb{E}[A]\right] \le \exp\left(-\frac{\delta^2}{2+\delta}\cdot\mathbb{E}[A]\right)$$

and setting $\delta = \frac{1+\epsilon}{X} - 1$, we get

$$\begin{aligned}\mathbb{P}[\mathcal{EV}_2(J,M)] \;&\le\; \exp\left(-\frac{(\frac{1+\epsilon-X}{X})^2}{2+\frac{1+\epsilon}{X}-1}\cdot\mathbb{E}[\hat{n}^J_M]\right) \;=\; \exp\left(-\frac{(1+\epsilon-X)^2}{X^2(\frac{1+\epsilon}{X}+1)}\cdot Xc\log n\cdot\frac{n}{k}\right)\\ &=\; \exp\left(-\frac{(1+\epsilon-X)^2}{X+1+\epsilon}\cdot c\log n\cdot\frac{n}{k}\right) \;=\; \exp\left(-f(X)\cdot c\log n\cdot\frac{n}{k}\right),\end{aligned}$$

where $f(x) = \frac{(1+\epsilon-x)^2}{x+1+\epsilon}$. It is easily verifiable that $f(x)$ is monotone decreasing in the range $[0,1]$, attaining a minimum value of $\frac{\epsilon^2}{2+\epsilon}$, i.e, $f(x) \ge \epsilon^2/2+\epsilon$ for every $x \in [0,1]$. Therefore, we get

$$\begin{aligned}\mathbb{P}\left[\hat{n}^J_M > \frac{1+\epsilon}{X}\cdot\mathbb{E}[\hat{n}^J_M]\right] \qquad &\le \exp\left(-\frac{\epsilon^2}{2+\epsilon}\cdot c\log n\cdot\frac{n}{k}\right) \;\le\; n^{-c\epsilon^2/(2+\epsilon)}\\ &\le\; n^{-Z\epsilon^2/(2+\epsilon)} \;\le\; \frac{1}{n^{2+\lambda}},\end{aligned}$$

where the last inequality follows by Eq. (3). The lemma now follows by the union bound. ◀

Almost Optimal Algorithms for Token Collision in Anonymous Networks

Sirui Bai
State Key Laboratory for Novel Software Technology, Nanjing University, China

Xinyu Fu
State Key Laboratory for Novel Software Technology, Nanjing University, China

Xudong Wu
State Key Laboratory for Novel Software Technology, Nanjing University, China

Penghui Yao
State Key Laboratory for Novel Software Technology, Nanjing University, China
Hefei National Laboratory, China

Chaodong Zheng
State Key Laboratory for Novel Software Technology, Nanjing University, China

Abstract

In distributed systems, situations often arise where some nodes each holds a collection of *tokens*, and all nodes collectively need to determine whether all tokens are distinct. For example, if each token represents a logged-in user, the problem corresponds to checking whether there are duplicate logins. Similarly, if each token represents a data object or a timestamp, the problem corresponds to checking whether there are conflicting operations in distributed databases. In distributed computing theory, unique identifiers generation is also related to this problem: each node generates one token, which is its identifier, then a verification phase is needed to ensure that all identifiers are unique.

In this paper, we formalize and initiate the study of *token collision*. In this problem, a collection of k tokens, each represented by some length-L bit string, are distributed to n nodes of an *anonymous* CONGEST network in an arbitrary manner. The nodes need to determine whether there are tokens with an identical value. We present near optimal deterministic algorithms for the token collision problem with $\tilde{O}(D + k \cdot L/\log n)$ round complexity, where D denotes the network diameter. Besides high efficiency, the prior knowledge required by our algorithms is also limited. For completeness, we further present a near optimal randomized algorithm for token collision.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Token collision, anonymous networks, deterministic algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.4

Related Version *Full Version*: `https://arxiv.org/abs/2408.10519`

Funding This work is supported by the National Natural Science Foundation of China (Grant No. 62332009, 62172207, 12347104), the Department of Science and Technology of Jiangsu Province of China (Grant No. BK20211148) and Innovation Program for Quantum Science and Technology (Grant No. 2021ZD0302901).

1 Introduction

Imagine the following scenario: a group of servers is hosting an online-banking, online-gaming, or online-exam service; for security reasons, users are not allowed to log into multiple servers simultaneously. If we interpret each logged-in user as a *token*, the servers need to check whether all active tokens are distinct. Similar problems could also arise in distributed database systems. For example, in some distributed databases, optimistic concurrency control schemes are employed to increase concurrency and performance [17]. Motivation for

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 4; pp. 4:1–4:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

such schemes is the observation that system clients are unlikely to access the same object concurrently. Nonetheless, before the system commit clients' transactions, verification must be performed to ensure that all read and write operations are disjoint or that no operations occur at the same time, otherwise a rollback is necessary. In this setting, tokens represent data objects or timestamps [19, 20].

Apart from above practical scenarios, detecting colliding tokens is also important from a theoretical perspective as we can interpret identifiers as tokens. Specifically, it is well-known that a number of fundamental distributed computing tasks, such as coloring, leader election, bipartiteness testing, and planarity testing, are impossible to resolve deterministically in anonymous networks [3, 15, 21]. Hence, unique identifiers generation becomes an important primitive for anonymous networks, as it breaks the symmetry within the network, thus making the aforementioned tasks possible. Moreover, the lengths of these identifiers could affect the performance of corresponding algorithms; examples include renaming algorithms, Linial's classical n-to-Δ^2 coloring algorithm [21], recent deterministic network decomposition algorithms by Ghaffari et al. [11], and recent MST algorithm focusing on energy complexity [5]. A plausible approach for generating unique identifiers in anonymous networks is to employ randomness: for instance, each of the n nodes generates an identifier by sampling $\Theta(\log n)$ uniform random bits; by the birthday paradox, all identifiers are distinct with probability at least $1 - 1/n$. However, this is a Monte Carlo algorithm that is subject to error. If we seek a Las Vegas (i.e., zero-error) algorithm for generating unique identifiers in anonymous networks, a deterministic algorithm for detecting colliding identifiers (i.e., tokens) is necessary: nodes repeatedly run a (Monte Carlo) randomized identifiers generation algorithm and use a deterministic algorithm to check whether the generated identifiers are unique.

Despite the various applications for *token collision*, somewhat surprisingly, this problem has not been explicitly studied in the context of distributed computing to the best of our knowledge. In this paper, we initiate the study of this generic distributed computing task, and we begin by giving a definition for it:

▶ **Definition 1 (Token Collision).** *Assume that there are k tokens, each having a value represented by a length-L binary string. Consider a distributed system consisting of n nodes. The k tokens are divided into n collections, some of which may be empty. Each of the n nodes is assigned one collection as input. In the* token collision *problem, the nodes need to determine whether there are tokens with identical value.*

We focus on understanding the time complexity of token collision in anonymous networks. The reason for considering anonymous networks is two-fold. First, if nodes in a distributed system already have unique identifiers, then token collision (or almost any distributed computing task) could be resolved by first electing the node with the smallest identifier as the leader, then aggregate necessary information to the leader, and finally let the leader locally compute the result and disseminate the result to the rest of the network.[1] Second, anonymous networks also arise in real-world scenarios. For example, the nodes in a distributed system (e.g., sensor networks) may be indistinguishable since they are fabricated in a large-scale industrial process, in which equipping every node with a unique identifier is not economically feasible (e.g., MAC addresses are not necessarily unique nowadays). In other cases nodes may not wish to reveal their identities out of privacy or security concerns.

[1] Nonetheless, our lower bounds for token collision hold even for named networks, and our algorithms nearly match these lower bounds. Thus, anonymity does not make token collision harder.

We consider standard CONGEST model [21] in distributed computing. A CONGEST network is described by a graph $G = (V, E)$ with $|V| = n$ nodes being processors with unlimited computational power (we do not exploit this ability in this paper) and edges being communication channels with bounded bandwidth. Specifically, we assume that any message sent through a channel cannot exceed $\Theta(\log n)$ bits. To simplify presentation, we often use $B = \Theta(\log n)$ to denote this bandwidth limitation. Processors exchange messages synchronously, round-by-round along the channels. When proving impossibility results for the token collision problem, we also consider an alternate model known as the LOCAL model [18], where communication channels have unbounded bandwidth.

1.1 Results and contribution

Deterministic scenario. We first consider the case where the tokens are not too large – particularly, every token can fit into one message. In this scenario, we offer a deterministic algorithm that works so long as every node knows the exact value of n or k.

▶ **Theorem 2** (**Deterministic Upper Bound, Part 1**). *In an n-node anonymous CONGEST network with diameter D, for any instance of the token collision problem in which k tokens are encoded by length-L bit strings, if every node knows the exact value of n or k, then there exists an $O(D + k \cdot L/\log n)$-round deterministic algorithm when $L = O(\log n)$.*

The above theorem implies when $L = \Theta(\log n)$, token collision can be resolved within $O(D+k)$ rounds. As a result, one could easily derive a Las Vegas unique identifiers generation algorithm with $O(n)$ expected runtime. On the other hand, for problem instances where tokens are small – $L = o(\log n)$ in particular, the runtime of the algorithm is $O(D) + o(k)$.

At a high level, our algorithm tries to find the token(s) that have the global minimum value and selects the node(s) that own(s) such token(s) as leader(s). Critically, our algorithm may elect multiple nodes as leaders, so it does not solve the leader election problem. (In fact, leader election cannot be solved deterministically in our setting.) Nevertheless, alongside this election process, BFS-trees will be built with these leaders being the roots, thus the network graph becomes a forest logically. Then, by convergecasting [4] tokens within each tree and computing the size of each tree, root nodes can correctly determine the result. Similar ideas have been used in the design of Las Vegas leader election algorithms, but the analysis of our deterministic algorithm is more challenging, see Section 1.2 for more discussion.

We then extend our algorithm to the scenario where each token cannot fit into one message, this could occur in applications like plagiarism checking in which each token is a text segment. In this case, a simple solution is to divide each token into $\Theta(L/\log n)$ parts, and use multiple rounds to simulate one round of our above algorithm. However, the resulting algorithm would have a round complexity of $O((D+k) \cdot L/\log n)$, which is too large. Instead, we devise a variant that uses pipelining techniques and extend the analysis accordingly. The following theorem states the time complexity of this variant.

▶ **Theorem 3** (**Deterministic Upper Bound, Part 2**). *In an n-node anonymous CONGEST network with diameter D, for any instance of the token collision problem in which k tokens are encoded by length-L bit strings, if every node knows the exact value of n or k, then there exists an $O(D \cdot \max\{(\log(L/\log n))/\log n, 1\} + k \cdot L/\log n)$-round deterministic algorithm when $L = \omega(\log n)$.*

To complement the algorithmic results, we have also established a lower bound on the round complexity of the token collision problem.

▶ **Theorem 4 (Deterministic Lower Bound).** *In an n-node anonymous CONGEST network with diameter D, there are instances of the token collision problem in which k tokens are encoded by length-L bit strings such that, any deterministic algorithm requires $\Omega(D + k \cdot (L - \log k + 1)/\log n)$ rounds to solve it when $2^L \geq k$.*

It is easy to verify, if $L \geq (1 + \delta) \log k$ for some constant $\delta > 0$, then the round complexity of our algorithm is tight when $L = O(\log n)$, and near-optimal (within multiplicative $\log(L/\log n)/\log n$ factor) when $L = \omega(\log n)$. We also note that the assumption of $2^L \geq k$ is without loss of generality, as otherwise collision must occur. To obtain the lower bound, we reduce a variant of the set-disjointness problem in the study of two-party communication complexity to the token collision problem.

Another advantage of our algorithm is that it requires little prior knowledge. Beside input tokens, each node only needs to know the value of n *or* the value of k. In particular, nodes do not need to know the network diameter D. In fact, we can prove via an indistinguishability argument that without any global knowledge, deterministic token collision detection is impossible. (Nonetheless, what is the minimal prior knowledge required remains to be an interesting open question.)

▶ **Theorem 5 (Impossibility Result).** *In the anonymous LOCAL model, if every node has no knowledge regarding the network graph except being able to count and communicate over adjacent links locally, and if every node also has no knowledge regarding the tokens except the ones given as local input, then there is no deterministic algorithm that solves the token collision problem.*

Randomized scenario. We also investigate the randomized round complexity of token collision. At the upper bound side, selecting a unique leader can be easily achieved with desirable probability when randomness is allowed. Then a random hash function is employed to reduce the bit-length of tokens if they are too large, without increasing the probability of collision significantly in case tokens are distinct. Finally, with a convergecast process similar to the deterministic algorithm, the unique leader can collect all tokens and determine whether there are collisions. At the lower bound side, we again employ the strategy of reduction, and utilize existing results on the hardness of randomized set-disjointness to obtain the desired result. Our findings for the randomized scenario are summarized below; see Appendix B for more details. Note that in contrast with the deterministic setting, the length of the token L no longer appears in the lower bound, and L's impact on the upper bound is also limited.

▶ **Theorem 6 (Randomized Upper and Lower Bound).** *Consider an n-node anonymous CONGEST network with diameter D. For any instance of the token collision problem with k tokens, if every node knows the exact value of n or k, then there exists a randomized algorithm that solves it in $O(D \cdot \max\{(\log(\log k/\log n))/\log n, 1\} + k \cdot \log k/\log n + L/\log n)$ rounds with probability at least $1 - 1/k$. On the other hand, assuming $2^L \geq k$, there are instances of the token collision problem with k tokens such that any randomized algorithm that succeeds with probability at least $2/3$ requires $\Omega(D + k/\log n)$ rounds.*

1.2 Related work and discussion

Though token collision has not been explicitly studied in distributed computing, similar problems have been investigated elsewhere. For example, element distinctness, which decides whether a given set of elements are distinct, has been extensively studied in the context of query complexity. Specifically, linear lower bounds were proved for deterministic and randomized algorithms [7, 12]. A sublinear quantum algorithm was proposed by Buhrman et al. [9], which applies $O(n^{3/4})$ quantum queries. The upper bound was later improved to $O(n^{2/3})$ by Ambainis using quantum walk [2], and matched the lower bound given by Aaronson and Shi [1]. To the best of our knowledge, this paper is the first one that studies token collision in classical distributed computing models, focusing on round complexity.

Token collision is also related to leader election, a classical and fundamental distributed computing primitive, in several aspects.

On the one hand, the design of Las Vegas leader election algorithms [14, 23, 10] share similar ideas with ours. In those algorithms, usually nodes first randomly generate identifiers, then the node with the smallest identifier is elected as the leader if that identifier is owned by a single node, otherwise the process restarts. As can be seen, the problem of checking whether the smallest identifier is unique is a variant of the token collision problem. Indeed, the routine developed by Tel in [23] for this checking procedure is very similar to our algorithm. Nonetheless, the analysis in our setting is more involved: in the context of Las Vegas leader election, restart when the smallest identifier is unique is fine (i.e., false negative is fine), yet in our context this is unacceptable. In fact, proving such false negative will not occur is highly non-trivial (see Lemma 12 in Section 4). Moreover, our algorithm can handle the scenario that tokens are of arbitrary size, making it more generic.

On the other hand, as mentioned earlier, with a unique leader almost any distributed computing problem can be solved. This observation raises the question that whether defining and studying token collision is necessary. We believe the answer is positive. First, the leader election approach is not necessarily better. Taking the unique identifiers generation problem as an example, the approach of “first elect a leader and then let the leader aggregate and check whether the generated identifiers are unique” share same round complexity with our algorithm. Second, and more importantly, in situations where leader election is infeasible (e.g., deterministic leader election in anonymous networks, randomized leader election that always terminates in anonymous rings of unknown size) [24], our algorithm still works with deterministic correctness and time complexity guarantees.

2 Preliminary

In this section, we briefly introduce some known results on the communication complexity of the set-disjointness problem as it is used in our lower bound proof.

Communication complexity was introduced by Yao [25], which is nowadays a versatile method to prove lower bounds in distributed computing. In the two-party communication complexity model, two players Alice and Bob, respectively, receive $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ as input and need to compute $f(x, y)$, where $f : \mathcal{X} \times \mathcal{Y} \to \mathcal{Z}$ is a two-argument function. The communication complexity of f is the minimum number of bits Alice and Bob need to exchange to compute $f(x, y)$ for any input x and y.

The problem of set-disjointness (denoted as DISJ) is one of the most well-studied problems in communication complexity [16, 22, 6], where Alice and Bob are given a set, respectively, and they need to decide whether their sets are disjoint. In this paper, we are interested in a variant of set-disjointness: $\mathrm{DISJ}_q^p : \binom{[p]}{q} \times \binom{[p]}{q} \to \{0, 1\}$. Alice is given a set $S \subseteq [p]$ and Bob is given a set $T \subseteq [p]$, where $|S| = |T| = q$. They aim to determine whether the two sets are disjoint, that is, $\mathrm{DISJ}_q^p(S, T) = 1$ iff $S \cap T = \emptyset$. The communication complexity of DISJ_q^p is established by Håstad and Widgerson [13].

▶ **Fact 7** ([13]). *For every $q \le p/2$, $D(\mathrm{DISJ}_q^p) = \Omega(\log \binom{p}{q})$ and $R_{1/3}(\mathrm{DISJ}_q^p) = \Omega(q)$. Here, $D(\mathrm{DISJ}_q^p)$ denotes the deterministic communication complexity of problem DISJ_q^p, and $R_{1/3}(\mathrm{DISJ}_q^p)$ denotes the randomized communication complexity of problem DISJ_q^p with the probability of error being at most $1/3$.*

3 The Deterministic Algorithm

In this section, we focus on the most common scenario where each token can be fitted into one message (that is, $L = O(\log n)$). We will extend our algorithm to other settings later.

Broadly speaking, our algorithm can be divided into two parts: the first part concerns with building rooted BFS-trees, while the second part concerns with calculating the size of the BFS-trees and aggregating tokens at the roots for decision-making. Although the high-level idea of our algorithm is not complicated, implementing it correctly and efficiently is non-trivial, especially in the setting where nodes only have limited global knowledge.

We now describe the algorithm in detail. (Complete pseudocode of the algorithm is provided in Appendix A.)

Build BFS-tree(s). Initially, each node v sets its identifier to be the smallest token it received as input, or a special symbol if v received no token. Then, it attempts to construct a BFS-tree rooted at itself by broadcasting its identifier to its neighbors in each round. Whenever v receives a smaller identifier from some neighbor u, it updates its identifier to match that of neighbor u. Moreover, it designates u as its parent and sends a notification to its parent in all subsequent rounds. As a result, whenever v changes its identifier to that of some neighbor u's, node v is appending the BFS-tree rooted at itself to the BFS-tree that includes u. We note that each node v uses a variable $\mathtt{rid}_v$ to store its identifier. Intuitively, $\mathtt{rid}_v$ stores the root's identifier of the BFS-tree that v belongs to.[2] We also note that each node v uses an integer $\mathtt{p}_v \in [\Delta_v]$ to store its parent, where Δ_v is the degree of v. That is, each node v locally labels each incident edge with a unique integer in $[\Delta_v]$, and uses the edge label as its local identity for the node at the other endpoint of the edge.

When node v discovers that all of its neighbors share the same identifier as itself, it attempts to ascertain whether the BFS-tree rooted at itself is fully constructed. To this end, note that the BFS-tree rooted at node v consists of node v and the BFS-trees rooted at its children. Therefore, our algorithm's criterion for node v to confirm that the BFS-tree rooted at itself is fully constructed is: all v's neighbors share v's identifier and the BFS-trees rooted at its children are fully constructed. To implement this idea, each node v stores a boolean variable $\mathtt{f}_v$ to indicate whether the BFS-tree construction process is completed, and $\mathtt{f}_v$ is sent to v's parent in each round. Initially $\mathtt{f}_v$ is $false$, and $\mathtt{f}_v$ becomes $true$ if: (1) all v's neighbors share identical identifier as v; and (2) each child u of v has $\mathtt{f}_u = true$ or v has no children (that is, v is a leaf node).

Lastly, if node v determines that the BFS-tree rooted at itself is fully constructed and it does not have a parent, then it broadcasts a termination signal to its neighbors once and stops the BFS-tree building procedure. The node v will then proceed to the second stage of the algorithm. On the other hand, whenever a node receives a termination signal, it also stops its BFS-tree building procedure, broadcasts this signal to all neighbors once, and then proceed to the second stage of the algorithm. During algorithm execution, each node v uses a boolean variable $\mathtt{build}_v$ to maintain this signal: $\mathtt{build}_v$ is initially $true$, and will be set to $false$ when v's BFS-tree building procedure is done. Notice that if there are multiple BFS-trees being constructed simultaneously, after the first one completes, the flooding mechanism of the termination signal may stop the remaining ones from being completed. Nonetheless, such disruption is fine: the existence of multiple BFS-trees implies there are token collisions, and our algorithm can correctly detect this later.

[2] This is merely an "intuition" and not always true during BFS-tree construction, as identifiers are propagating gradually and tree shape may change frequently.

Detect token collision. The token-collision detection procedure has two main tasks: compute BFS-tree's size and aggregate tokens. A node only starts this procedure if itself and all its neighbors have terminated the BFS-tree construction procedure.

To determine the size of the BFS-tree rooted at itself, node v first identifies its children. It filters out the neighbors that have the same identifier as itself and have designated node v as their parent. Node v uses $\texttt{chi}_v$ to store this set of children. If node v finds that all of its children in $\texttt{chi}_v$ have already computed the size of their respective BFS-trees, then node v can calculate the size of the BFS-tree rooted at itself. This is done by summing up the sizes of the BFS-trees rooted at its children and adding one to account for node v itself. During algorithm execution, each node v uses $\texttt{cnt}_v$ to track the size of the BFS-tree rooted at itself. Initially $\texttt{cnt}_v$ is set to a special symbol $\perp$. Later when v has finished counting, $\texttt{cnt}_v$ becomes an integer.

It remains to aggregate the tokens. In each round, after node v receives all messages (which may include tokens from its children), if v has a parent u and the token list of v is not empty, then v ejects one token from its token list and sends that token to u in the next round. Node v also needs to tell its parent u whether all tokens in the BFS-tree rooted at v has already been transferred to u. To this end, in each round, after v has received all messages, if the token list of v is empty and every child of v indicates all tokens have already been transferred to v, then v concludes that all tokens in the BFS-tree rooted at itself has already been transferred to its parent. It will inform its parent u about this in the next round. During algorithm execution, each node v uses $\boldsymbol{x}^v$ to store its token list and uses $\texttt{ele}_v$ to denote the token that v intends to send to its parent. We note that $\texttt{ele}_v$ is set to $\perp$ when v's token list is empty (that is, $|\boldsymbol{x}^v| = 0$) and every child w of v indicates all tokens in the subtree rooted at w has been transferred to v (that is, w tells v $\texttt{ele}_w =\perp$); and $\texttt{ele}_v$ is set to $\top$ when v's token list is empty but some child w of v indicates there still are tokens pending to be transferred to v (that is, w tells v $\texttt{ele}_w = \top$).

If node v does not have a parent, it must be the root of some entire BFS-tree and is responsible for deciding the result of token collision. To this end, once v has obtained the size of the BFS-tree rooted at itself and all its children signal that the tokens have been transferred to v, it determines the result of token collision as follows. In the case that all nodes know the exact value of n, if the size of the BFS-tree rooted at v equals n and no token collision is found in the token list of v, then v can confirm the non-existence of token collision. Otherwise, a token collision must exist. In the case that all nodes know the exact value of k, if the size of the token list of v equals k and no token collision is found, then v can confirm the non-existence of token collision. Otherwise, a token collision must exist. Node v uses a boolean variable $\texttt{res}_v$ to store the result. It will broadcast the result to all its neighbors once in the next round and then halt. Upon receiving the result, every node also broadcasts the result to its neighbors once in the next round and then halts.

4 Analysis of the Deterministic Algorithm

In this section, we show the correctness of our algorithm and analyze its running time. Omitted proofs are provided in the full version of the paper.

4.1 Correctness

We begin with the correctness guarantees: if all nodes know n or k, then all nodes return an identical and correct result on whether there are collisions among the k tokens.

To prove the above claim, we first argue the correctness of our BFS-tree construction procedure. Specifically, we intend to show that our algorithm always maintains a directed forest $G' = (V, E')$ where a directed edge $(v, u) \in E'$ if node v assigns node u as its parent. To this end, we introduce the notion of *identifier-induced graph*.

▶ **Definition 8 (Identifier-induced Graph).** *At the end of any round, define directed graph $G' = (V, E')$ as the* identifier-induced graph *in the following way: V is the node set of the network graph, and a directed edge $(v, u) \in E'$ if v assigns u as its parent.*

To show that the identifier-induced graph is a forest, we begin with the following observation. Recall that each node v uses variable `rid` to store its identifier. Intuitively, this lemma holds since node v only updates the value of $\texttt{rid}_v$ to the value of $\texttt{rid}_u$ and sets its parent pointer to u when v receives $\texttt{rid}_u$ from some neighbor u with $\texttt{rid}_u < \texttt{rid}_v$.

▶ **Lemma 9.** *At the end of any round, for any directed path in the identifier-induced graph, the identifiers of the nodes along the directed path are non-increasing.*

Then, we can show the identifier-induced graph is a directed forest containing one or more rooted trees. To prove the lemma, the key is to show there are no directed cycles in the identifier-induced graph, which can be done by induction on round number.

▶ **Lemma 10.** *At the end of any round, the identifier-induced graph is a directed forest in which every weakly connected component is a rooted tree. In particular, in each tree, the unique node with no parent is the root of that tree.*[3]

Next, we show an important property regarding the rooted trees in the identifier-induced graph. Intuitively, it states that within each such tree, nodes may have different identifiers, but for any subtree within the tree, the nodes that have identical identifiers with the root of the subtree are connected and are at the "top" of the subtree.

▶ **Lemma 11.** *At the end of any round, for any node r, within the subtree rooted at node r in the identifier-induced graph, the subgraph induced by the nodes having identical identifier with node r is also a tree rooted at node r.*

The following key lemma shows that when there are no token collisions, the BFS-tree building procedure constructs a single rooted tree containing all nodes. Though the claim seems straightforward, proving it rigorously turns out to be highly non-trivial.

▶ **Lemma 12.** *If there are no token collisions, then after all nodes quit the BFS-TREE-BUILDING procedure – that is, after each node v sets $\texttt{build}_v = false$, the identifier-induced graph contains a single tree rooted at the node having the minimum token as input, and all nodes in that tree have identical identifier.*

Proof sketch. Throughout the proof, assume there are no token collisions. For each node v, let $\texttt{id}_v$ denote the minimum token that v received as input. Let $v_{\min}$ denote the unique node having the smallest input token. For any two nodes $u, v \in V$, let $\texttt{dist}(u, v)$ denote the distance between u and v in the network graph G. Define $d = \max_{v \in V} \texttt{dist}(v, v_{\min})$. For any node v, let d_v denote the distance between node v and the nearest node u with $\texttt{id}_u$ smaller than v. That is, $d_v = \min_{u \in V, \texttt{id}_u < \texttt{id}_v} \texttt{dist}(u, v)$. We set $d_{v_{\min}} = +\infty$.

We make the following three claims and prove their correctness via induction on rounds. These claims highlight the key properties our BFS-tree building procedure can enforce.

[3] A weakly connected component of a directed graph is a connected component of the graph when ignoring edge directions.

1. No node quits the BFS-tree building procedure within d rounds. Formally, for any $0 \le i \le d$, each node v has $\texttt{build}_v = \mathit{true}$ by the end of round i.
2. For any node v, any $0 \le i \le d$, let node $\hat{u}$ be the unique node that has minimum $\texttt{id}$ among all nodes u with $\texttt{dist}(u, v) \le i$. At the end of round i, we have $\texttt{rid}_v = \texttt{id}_{\hat{u}}$. Formally, at the end of round i, it holds that $\texttt{rid}_v = \min_{u \in V, \texttt{dist}(u,v) \le i} \texttt{id}_u$.
3. For any node v, any $0 \le i \le d$, at the end of round i, one of the following cases holds.
   - Case I: $d_v > 2i$. The nodes with distance at most i to v form a height-i tree rooted at v in the identifier-induced graph. Moreover, for any node u with distance i to v, for any node w on the directed path from u to v in the rooted tree, it holds that $\texttt{f}_w = \mathit{false}$.
   - Case II: $i < d_v \le 2i$. There is a tree rooted at v in the identifier-induced graph. Let node w denote the unique node with distance d_v to node v that has the smallest $\texttt{id}$. There exists a node u with $\texttt{dist}(u, v) = d_v - i - 1$ and $\texttt{dist}(u, w) = i + 1$ such that $\texttt{f}$ is false for all nodes along the length-$(d_v - i - 1)$ directed path from u to v in the identifier-induced graph.
   - Case III: $d_v \le i$. Node v has a parent in the identifier-induced graph.

The claims above easily lead to the lemma. By Item 1, no node will quite the BFS-Tree-Building procedure within d rounds. By Item 2, at the end of round d, all nodes have identical $\texttt{rid}$, which is $\texttt{id}_{v_{min}}$. By Case I of Item 3, at the end of round d, all nodes form a single tree rooted at v_{min} in the identifier-induced graph. Moreover, no node will ever change parent or $\texttt{rid}$ later. ◀

We now proceed to argue the correctness of the token aggregation process, which will lead to the correctness of our entire algorithm. Recalling Lemma 11, we begin by defining *identifier-induced subtree* to facilitate presentation.

▶ **Definition 13 (Identifier-induced Subtree).** *At the end of any round, for any node r, within the subtree rooted at r in the identifier-induced graph, call the subtree induced by the nodes that have identical identifier with r as the* identifier-induced subtree rooted at r.

Our first lemma regarding the correctness of the token aggregation process states that, informally, every root r in the identifier-induced graph correctly computes the size of the identifier-induced subtree rooted at r if it outputs a decision for the token collision problem.

▶ **Lemma 14.** *Assume that in some round i, node v runs procedure Token-Collision-Detection and within that procedure updates $\texttt{res}_v$ for the first time (so that after the update $\texttt{res}_v \neq \perp$), then by the end of round i, the value of $\texttt{cnt}_v$ equals the size of the identifier-induced subtree rooted at v.*

Proof sketch. Notice that in our algorithm, only a root node in the identifier-induced graph can update its $\texttt{res}$ within procedure Token-Collision-Detection, so v must be a root node in the identifier-induced graph by the end of round i. Let $T_{v,i}$ be the identifier-induced subtree rooted at v by the end of round i. We need to show that $\texttt{cnt}_v$ equals the size of $T_{v,i}$ by the end of round i.

To prove the above result, we make the following claim and prove it via an induction on round number: for any node u, if i_u is the first round in which u updates $\texttt{cnt}_u$ to some non-$\perp$ value, then $\texttt{cnt}_u$ equals the size of T_{u,i_u} by the end of round i_u, where T_{u,i_u} is the identifier-induced subtree rooted at u by the end of round i_u. Moreover, by the end of any round $i' > i_u$, $\texttt{cnt}_u$ remains unchanged and $T_{u,i'}$ is identical to T_{u,i_u}.

With the above claim, the lemma is easy to obtain. Assume i_v is the first round in which v updates $\texttt{cnt}_v$ to some non-$\perp$ value, then $\texttt{cnt}_v$ equals the size of T_{v,i_v} by the end of round i_v. Later, at the end of round i, when v updates $\texttt{res}$ to some non-$\perp$ value in procedure Token-Collision-Detection, $\texttt{cnt}_v$'s value remains unchanged and is $|T_{v,i_v}| = |T_{v,i}|$. ◀

Our second lemma regarding the correctness of the token aggregation process states that, informally, every root r in the identifier-induced graph correctly collects the tokens within the identifier-induced subtree rooted at r if it outputs a decision for the token collision problem. The high-level strategy for proving this lemma is similar to the proof of Lemma 14, but the details are more involved as the convergecast process is more complicated.

▶ **Lemma 15.** *Assume that in some round i, node v runs procedure* TOKEN-COLLISION-DETECTION *and within that procedure updates* $\texttt{res}_v$ *for the first time (so that after the update* $\texttt{res}_v \neq \perp$*), then by the end of round i, node v collects each token owned by the nodes within the identifier-induced subtree rooted at v exactly once.*

At this point, we are ready to show the correctness of our algorithm.

▶ **Lemma 16.** *After all nodes halt (that is,* $\texttt{res} \neq \perp$*), they return the correct result.*

Proof. Notice that by algorithm description and Lemma 10, only root nodes in the identifier-induced graph can generate $\texttt{res} \neq \perp$, other nodes can only passively adopt $\texttt{res} \neq \perp$ from neighbors. So, let v be an arbitrary node that generates $\texttt{res} \neq \perp$ and assume this happens in round i_v, then v must be a root in the identifier-induced graph by the end of round i_v. Let T_{v,i_v} denote the identifier-induced subtree rooted at v by the end of round i_v.

First consider the case v sets $\texttt{res} = \mathit{true}$ (that is, there are no token collisions). Then by Lemma 14 and Lemma 15, v has correctly collected the tokens in T_{v,i_v} and correctly counted the size of T_{v,i_v} by the end of round i_v, implying that T_{v,i_v} contains all input tokens and there are no collisions among input tokens; that is, the result v generated is correct. Moreover, since T_{v,i_v} is of size n, it must be the only tree in the identifier-induced graph. As a result, all nodes other than v will only passively adopt the result generated by v, implying that all nodes return identical result.

Next, consider the case v sets $\texttt{res} = \mathit{false}$ (that is, there are token collisions). Then, again, by Lemma 14 and Lemma 15, v has correctly collected the tokens in T_{v,i_v} and correctly counted the size of T_{v,i_v} by the end of round i_v. Since v sets $\texttt{res}$ to false, there are three possible reasons:

- All nodes know n but $|T_{v,i_v}| \neq n$. Recall Lemma 12, which states that if there are no token collisions, then there is only one tree in the identifier-induced graph that contains all nodes and all tokens. Hence, if $|T_{v,i_v}| \neq n$, then there are indeed token collisions. Furthermore, for any other root node u in the identifier-induced graph that also generates an $\texttt{res} \neq \perp$ by the end of some round i_u, node u must have also found $|T_{u,i_u}| \neq n$ and set $\texttt{res} = \mathit{false}$. Therefore, in this case, all nodes output the correct result.
- All nodes know k but the number of tokens v has collected is not k. By a similar argument as in the first case, we can conclude that all nodes output the correct result.
- Node v finds collisions among the tokens it has collected. In this case, v's decision to generate $\texttt{res} = \mathit{false}$ is obviously correct. Moreover, for any other root node u in the identifier-induced graph that also generates an $\texttt{res} \neq \perp$, that $\texttt{res}$ must be false, as the identifier-induced subtree rooted at u will not contain all n nodes or all k tokens.

This completes the proof of the lemma. ◀

4.2 Complexity

We now proceed to analyze the time complexity of the algorithm. The first lemma states that any identifier-induced subtree rooted at some node that has a global minimum token as input has limited height – particularly, $O(D)$. Moreover, each such node is a root in identifier-induced graph.

▶ **Lemma 17.** *Let v be a node having a minimum token as input. Then at the end of any round, v is a root in the identifier-induced graph, and the identifier-induced subtree rooted at v has height $O(D)$.*

With Lemma 17, we argue that all nodes finish BFS-Tree-Building in $O(D)$ rounds.

▶ **Lemma 18.** *After $O(D)$ rounds, every node v quits BFS-Tree-Building (that is, $\texttt{build}_v = false$).*

Proof. To prove the lemma, we only need to show that some node will quit BFS-Tree-Building within $O(D)$ rounds. This is because the flooding mechanism of a $false$-valued `build` variable ensures, once a node v sets $\texttt{build}_v = false$, all other nodes will set `build` to $false$ within (at most) another D rounds.

If some node quits BFS-Tree-Building within D rounds then we are done, so assume that this is not true. Then, by the end of round D, global minimum token's value is known by every node. Particularly, by the end of round D, each node has an `rid` with a value equals to some global minimum token. In other words, by the end of round D, each node is in some identifier-induced subtree rooted at some node that has a global minimum token as input. Moreover, no node will change its `rid` or parent ever since. By Lemma 17, any tree rooted at some node that has a global minimum token as input has $O(D)$ height.

Now, by our algorithm, starting from round $D + 1$, nodes within any such tree will start setting `f` to *true* from leaves to root. Since the height of any such tree is $O(D)$, after $O(D)$ rounds, either some node already sets `build` to $false$ and quits BFS-Tree-Building, or some root of such tree sets `build` to $false$ and quits BFS-Tree-Building. In both cases, some node quits BFS-Tree-Building within $O(D)$ rounds since the start of execution. ◀

The next lemma states the time complexity of our algorithm.

▶ **Lemma 19.** *After $O(D + k)$ rounds, every node v halts (that is, v returns $\texttt{res} \neq \perp$).*

Proof sketch. Recall that our algorithm guarantees that if one node generates an $\texttt{res} \neq \perp$ and then halts, then this `res` is broadcast to all other nodes. Hence, all other nodes will halt within another D rounds. As a result, to prove the lemma, we show that some node will halt within $O(D + k)$ rounds. To this end, we show that after all nodes quit BFS-Tree-Building which happens within $O(D)$ rounds (by Lemma 18), there exists a tree in the identifier-induced graph of height $O(D)$ (by Lemma 17), and the convergecast process inside this tree take $O(D + k)$ rounds. ◀

4.3 Proof of the main theorem

We now prove Theorem 2. When $L = \Theta(\log n)$ – meaning that each message can fit at most a constant number of tokens, by Lemma 16 and Lemma 19, the theorem is immediate.

When $L = o(\log n)$, to prove the theorem, we make a small modification to our algorithm: in the convergecast process, whenever a node forwards tokens to its parent, it packs as many tokens in a message as possible (particularly, $\Theta((\log n)/L)$ tokens in a message). Intuitively, this means that our algorithm is convergecasting $\Theta(kL/\log n)$ "packed tokens" each of size $\Theta(\log n)$, and each of these "packed tokens" contains $\Theta((\log n)/L)$ real tokens. Hence, the total runtime of our algorithm is still $O(D + kL/\log n)$ rounds.

The above argument is valid if, for every node that has some token(s) as input, that node receives at least $\Theta((\log n)/L)$ tokens. If some node only receives $o((\log n)/L)$ tokens as input (e.g., only one token), then a more careful analysis is required. Specifically, assume that there are x nodes that each receives $o((\log n)/L)$ tokens as input, call these nodes V_x, and the nodes in V_x in total have k_x tokens. So, there are $n-x$ nodes that each receives at least $\Theta((\log n)/L)$ tokens as input, call these nodes $V_{\overline{x}}$, and the nodes in $V_{\overline{x}}$ in total have $k-k_x$ tokens. Imagine a process in which we first aggregate the tokens owned by $V_{\overline{x}}$, and then aggregate the tokens owned by V_x. By the above analysis, aggregating the tokens owned by $V_{\overline{x}}$ takes $O(D+(k-k_x)L/\log n)$ rounds. On the other hand, for each token owned by some node in V_x, within $O(D)$ rounds, it either reaches the root, or arrives at a node that has at least $\Theta((\log n)/L)$ tokens pending to be sent. Effectively, this means that starting from the round we process the tokens owned by V_x, in $O(D)$ rounds, we again arrive at a scenario in which each node that has pending tokens to send has at least $\Theta((\log n)/L)$ tokens in its token list. As a result, these k_x tokens owned by V_x will all reach the root within $O(D+k_xL/\log n)$ rounds. Note that our modified algorithm cannot be slower than the imagined process, so the runtime of our modified algorithm when $L=o(\log n)$ is $O(D+kL/\log n)$.

5 Generalizing the Deterministic Algorithm when Tokens are Large

When tokens are large, $L=\omega(\log n)$ in particular, the time complexity of the BFS-tree building process and the token aggregation process are both affected. As mentioned in Section 1, we can apply the simple strategy of using $L/\log n$ rounds to simulate one round of our algorithm (as a token can be transferred in $L/\log n$ rounds), but the resulting algorithm would be too slow. Instead, in this section, we introduce and analyze a variant of our algorithm that costs only $O(D\cdot\max\{\frac{\log(L/\log n)}{\log n},1\}+k\cdot\frac{L}{\log n})$ rounds when $L=\omega(\log n)$.

The high level framework of this variant is the same as the algorithm introduced in Section 3: first build BFS-tree(s) and then detect token collisions within the tree(s). In this section, we focus on introducing the process of building BFS-tree(s) as the latter component is almost identical with the original algorithm. (Complete pseudocode of this variant is provided in Appendix A.) For the ease of presentation, we use $B=\Theta(\log n)$ to denote the bandwidth of CONGEST networks throughout this section.

5.1 Algorithm description

We first explain the key idea that allows this variant to be faster than the simulation strategy. Recall that in the original BFS-tree building process, each node v needs to record the minimum token it has seen in $\texttt{rid}_v$, and this is done by exchanging tokens in their *entirety* with neighbors. However, a key observation is, the relative order of two binary strings can be determined by a *prefix* of the strings that includes the most significant bit where they differ. As a result, we can employ the strategy that identifiers are sent successively starting from the most significant bit. Whenever a node v finds a prefix from some neighbor u is strictly smaller than the prefix of its current identifier, v updates its identifier to match the prefix and designates u as its parent. Moreover, when v sends its updated identifier, it does not need to restart from the first bit; instead, v starts from the bit where the updated identifier differs from the previous identifier. Effectively, we obtain an efficient "*pipeline*" approach on identifier broadcasting that can speed up the BFS-tree building process.

Build BFS-tree(s). We now detail how to implement the above idea. Similar to the original algorithm, each node v attempts to construct a BFS-tree rooted at itself by broadcasting its identifier $\texttt{rid}_v$. Due to bandwidth limitation, each identifier is divided into multiple *pieces* so that one piece can fit into one message. Denote these pieces as $\texttt{rid}_v[1],\cdots,\texttt{rid}_v[\lceil L/B\rceil]$, where $\texttt{rid}_v[1]$ contains the B most significant bits while $\texttt{rid}_v[\lceil L/B\rceil]$ contains the B least significant bits. The BFS-tree building procedure contains multiple *iterations*, each of which contains $\Theta(\frac{\log(L/B)}{B})$ rounds. In each iteration, $\Theta(\frac{\log(L/B)}{B})$ identifier pieces are sent, along with the position of the first sent piece in $\texttt{rid}_v$ – we use $\texttt{sent}_v$ to denote this position. (Notice that sending $\texttt{sent}_v$ may require $\frac{\log(L/B)}{B}$ rounds when L is large, this is why each iteration may contain multiple rounds.) Each node v locally maintains an identifier prefix for each neighbor based on received pieces. Whenever v finds a prefix of some neighbor u is strictly smaller than the prefix of its current identifier, v updates its identifier to match the prefix and designates u as its parent. At this point, v should send the updated identifier to neighbors. Particularly, v starts with the first piece where the updated identifier differs from v's previous identifier. This implies v may send non-successive piece position, in which case each neighbor of v should abandon the old prefix of v and record the new one.

Node v waits until all neighbors and itself have sent complete identifiers. Then, if v finds that all neighbors share the same identifier as itself, it attempts to ascertain whether the BFS-tree rooted at itself is fully constructed. Similar to the original algorithm, each node v uses a boolean variable $\texttt{f}_v$ to indicate whether BFS-tree construction is completed. Initially $\texttt{f}_v$ is $false$, and $\texttt{f}_v$ becomes *true* if: (1) v and all its neighbors have sent complete identifiers; (2) v and all its neighbors have identical identifier; and (3) each child u of v has $\texttt{f}_u = \mathit{true}$ or v has no children.

Lastly, if node v determines that the BFS-tree rooted at itself is fully constructed and it does not have a parent, then it terminates the BFS-tree building procedure and broadcasts a termination signal to all neighbors once. The node will then proceed to the second stage of the algorithm. Any node receiving such a signal will also forward it to neighbors once, stop the BFS-tree building procedure, and proceed to the second stage of the algorithm.

5.2 Analysis

The analysis for the above generalized algorithm is similar to the analysis for the original algorithm. Most claims and lemmas can carry over with little or no modifications, so are the proofs for these claims and lemmas. Others, however, require non-trivial extension or adjustments. To avoid redundancy, we only state these claims and lemmas here and provide proofs that require noticeable extension or adjustments in the full paper.

Correctness. The definition for identifier-induced graph remains unchanged in the generalized setting, except that such graph is defined at the end of each iteration.

▶ **Definition 20** (Analogue of Definition 8). *At the end of any iteration, define directed graph $G' = (V, E')$ as the* identifier-induced graph *in the following way: V is the node set of the network graph, and a directed edge $(v, u) \in E'$ if v assigns u as its parent.*

Following lemma is an analogue of Lemma 9, its proof is almost identical to that of Lemma 9, with small adjustments to account for the fact that identifiers are sent in pieces.

▶ **Lemma 21** (Analogue of Lemma 9). *At the end of any iteration, for any directed path in the identifier-induced graph, the identifiers of the nodes along the directed path are non-increasing.*

With Lemma 21, analogues of Lemma 10 and Lemma 11 hold automatically.

▶ **Lemma 22** (Analogue of Lemma 10). *At the end of any iteration, the identifier-induced graph is a directed forest in which every weakly connected component is a rooted tree. In particular, in each tree, the unique node with no parent is the root of that tree.*

▶ **Lemma 23** (Analogue of Lemma 11). *At the end of any iteration, for any node r that has sent its complete identifier to neighbors (that is, $\texttt{sent}_r = \lceil L/B \rceil$), within the subtree rooted at node r in the identifier-induced graph, the subgraph induced by the nodes having identical identifier with node r is also a tree rooted at node r.*

Lemma 12 is critical for the original algorithm, which states that a single BFS tree containing all nodes will be built when there are no token collisions. In the generalized setting, this claim still holds, but the proof needs to be extended in a non-trivial fashion to deal with the complication introduced by the pipeline approach for sending identifiers.

▶ **Lemma 24** (Analogue of Lemma 12). *If there are no token collisions, then after all nodes quit the BFS-Tree-Building procedure, the identifier-induced graph contains a single tree rooted at the node having the minimum token as input, and all nodes in that tree have identical identifier.*

Much like the case of Definition 20, the definition for identifier-induced subtree remains largely unchanged in the generalized setting.

▶ **Definition 25** (Analogue of Definition 13). *At the end of any iteration, for any node r that has sent its complete identifier to neighbors (that is, $\texttt{sent}_r = \lceil L/B \rceil$), within the subtree rooted at r in the identifier-induced graph, call the subtree induced by the nodes having identical identifier with r as the* identifier-induced subtree rooted at r.

Lemma 14 and Lemma 15 (and their proofs) still hold in the generalized setting, as we utilize the mechanism in the original algorithm for counting tree size and aggregating tokens.

▶ **Lemma 26** (Analogue of Lemma 14). *Assume that in some iteration i, node v runs procedure Token-Collision-Detection and within that procedure updates $\texttt{res}_v$ for the first time (so that after the update $\texttt{res}_v \neq \perp$), then by the end of iteration i, the value of $\texttt{cnt}_v$ equals the size of the identifier-induced subtree rooted at v.*

▶ **Lemma 27** (Analogue of Lemma 15). *Assume that in some iteration i, node v runs procedure Token-Collision-Detection and within that procedure updates $\texttt{res}_v$ for the first time (so that after the update $\texttt{res}_v \neq \perp$), then by the end of iteration i, node v collects each token owned by the nodes within the identifier-induced subtree rooted at v exactly once.*

We conclude this part with the following lemma which shows the correctness of our generalized algorithm, its proof is essentially identical to that of Lemma 16.

▶ **Lemma 28** (Analogue of Lemma 16). *After all nodes halt (that is, $\texttt{res} \neq \perp$), they return identical and correct result.*

Complexity. We now analyze the round complexity of the generalized algorithm, focusing on the BFS-tree construction process. Firstly, an analogue of Lemma 17 can be established.

▶ **Lemma 29** (Analogue of Lemma 17). *Let v be a node having a minimum token as input. At the end of any iteration, if v has sent its complete identifier to neighbors (that is, $\texttt{sent}_v = \lceil L/B \rceil$), then v is a root in the identifier-induced graph, and the identifier-induced subtree rooted at v has height $O(D)$.*

The next lemma states the time consumption of the BFS-tree construction process, it highlights the advantage of using the pipelining approach over the simulation approach.

▶ **Lemma 30** (Analogue of Lemma 18). *After $O(D + \frac{L}{\log(L/\log n)})$ iterations, every node v quits BFS-Tree-Building (that is, $\texttt{build}_v = false$).*

The last lemma shows the total time complexity of the generalized algorithm.

▶ **Lemma 31** (Analogue of Lemma 19). *After $O(D \cdot \frac{\log(L/\log n)}{\log n} + k \cdot \frac{L}{\log n})$ rounds, every node v halts (that is, v returns $\texttt{res} \neq \perp$).*

Proof of the main theorem. Combine Lemma 28 and Lemma 31, Theorem 3 is immediate.

6 Impossibility Result and Lower Bound for Deterministic Algorithms

Impossibility result. Recall Theorem 5 which states that if each node has no knowledge about the network graph except being able to count and communicate over adjacent links, and if each node also has no knowledge regarding the tokens except the ones being given as input, then the token collision problem has no deterministic solution.

To obtain the above impossibility, the key intuition is: to solve the problem, nodes need to exchange their input tokens in some manner; but in the anonymous setting with no global knowledge regarding network graph or input tokens, whenever a node receives a token from some neighbor that collide with its own input, the node cannot reliably determine whether this token originates from itself or some other node, yet the correctness of any algorithm depends on being able to distinguish these two scenarios.

We now provide a complete proof. Note that our impossibility result is strong in that we can construct counterexamples for any network size $n \geq 3$.

Proof of Theorem 5. Assume that there is an algorithm $\mathcal{A}$ that solves the token collision problem in the considered setting. For any $n \geq 3$, we consider two problem instances. The first instance – henceforth called C_n – is a ring consisting of n nodes, denoted as $v_1, v_2, \cdots, v_n$. Each node in the network obtains one token as input. Particularly, for any $i \in [n]$, node v_i has a token with value i. The second instance – henceforth called C_{2n} – is a ring consisting of $2n$ nodes. To construct C_{2n}, we first build two paths. The first path contains n nodes, denoted as $v'_1, v'_2, \cdots, v'_n$; the second path also contains n nodes, denoted as $u_1, u_2, \cdots, u_n$. Then, we connect v'_n with u_1, and connect u_n with v'_1. At this point, we have a ring. Each node in C_{2n} obtains one token as input. Particularly, for any $i \in [n]$, node v'_i and node u_i each has a token with value i.

Clearly, token collisions exist in C_{2n} but not in C_n, yet we will prove that $\mathcal{A}$ outputs identical results in both instances, resulting in a contradiction. Specifically, call the execution of $\mathcal{A}$ on C_n as α and the execution of $\mathcal{A}$ on C_{2n} as β, we will prove by induction that by the end of every round, for any $i \in [n]$, the internal states of nodes v_i, v'_i and u_i are identical.

The base case, which is immediately after initialization (i.e., round 0), trivially holds.

Assume that the claim holds for all rounds up to the end of round $r \geq 0$, now consider round $r + 1$. Fix an arbitrary $i \in [n]$, by the induction hypothesis, nodes v_{i-1} and v'_{i-1} have identical states by the end of round r. Hence, in round $r+1$, the message (if any) v_{i-1} sends to v_i and the message (if any) v'_{i-1} sends to v'_i will be identical. Similarly, in round $r+1$, the message (if any) v_{i+1} sends to v_i and the message (if any) v'_{i+1} sends to v'_i will be identical. Also, notice that by the end of round r, by the induction hypothesis, v_i and v'_i have identical states. Hence, during round $r + 1$, the local views of v_i and v'_i are identical. In other words, for any $\hat{v} \in \{v_i, v'_i\}$, node $\hat{v}$ cannot distinguish whether it is in α or β. Therefore, by the end of round $r+1$, nodes v_i and v_i' have identical states. By a similar argument, we can show that by the end of round $r+1$, nodes v_i and u_i also have identical states. This completes the proof of the inductive step, hence proving the claim.

Since $\mathcal{A}$ solves the token collision problem, α and β both terminate. Moreover, due to the above claim, nodes in α and β output identical results, resulting in a contradiction. ◀

Deterministic lower bound. We reduce the set-disjointness problem to the token collision problem and obtain the following theorem. Theorem 4 is an immediate corollary of it (by setting $\mathrm{mincut}(G) = 1$).

▶ **Theorem 32.** *Recall the parameters n, k, L introduced in the definition of the token collision problem (that is, Definition 1). Consider a size-n CONGEST network $G = (V, E)$ with diameter D. Assuming $2^L \geq k$, any deterministic algorithm that solves the token collision problem takes $\Omega(D + \frac{k(L-\log k+1)}{\mathrm{mincut}(G)\cdot\log n})$ rounds. Here,* $\mathrm{mincut}(\mathrm{G})$ *denotes the mincut of G. That is,* $\mathrm{mincut}(G) = \min_{U\subset V} |\{(u,v) \in E \mid u \in U, v \in V \setminus U\}|$.

Proof. Let $(U, V \setminus U)$ be a partition of V that attains $\mathrm{mincut}(G)$. Let $u \in U$ and $v \in V \setminus U$ be a pair of farthest nodes between U and $V \setminus U$. Assume that u and v are assigned token sets S and T respectively, each containing $k/2$ tokens.

Notice that $\Omega(D)$ is a lower bound for the token collision problem, as the distance between u and v is $\Theta(D)$, and they need to communicate with each other to solve the problem.

On the other hand, recall the two-party communication model and the set-disjointness problem introduced in Section 2. Since $2^L \geq k$, by setting $p = 2^L$ and $q = k/2$, it holds $q \leq p/2$. Recall the bandwidth of the network is $B = \Theta(\log n)$. We claim, if there exists an r-round algorithm that deterministically solves token collision in the CONGEST model, then Alice and Bob can compute $\mathrm{DISJ}_q^p(S,T)$ by communicating at most $2rB \cdot \mathrm{mincut}(G)$ bits. Specifically, they can run the r-round algorithm by having Alice and Bob simulate nodes in U and $V \setminus U$ respectively. Communication between Alice and Bob is necessary only when messages (each of which is at most B bits) are exchanged between nodes in U and $V \setminus U$ in the simulation. Apply Fact 7, we have $2r \cdot \mathrm{mincut}(G) \cdot B = \Omega(\log \binom{p}{q})$, implying $r = \Omega(\frac{k(L-\log k+1)}{\mathrm{mincut}(G)\cdot B}) = \Omega(\frac{k(L-\log k+1)}{\mathrm{mincut}(G)\cdot\log n})$. ◀

References

1 Scott Aaronson and Yaoyun Shi. Quantum lower bounds for the collision and the element distinctness problems. *Journal of the ACM*, 51(4):595–605, 2004. doi:10.1145/1008731.1008735.

2 Andris Ambainis. Quantum Walk Algorithm for Element Distinctness. *SIAM Journal on Computing*, 37(1):210–239, 2007. doi:10.1137/S0097539705447311.

3 Dana Angluin. Local and Global Properties in Networks of Processors (Extended Abstract). In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, STOC, pages 82–93. ACM, 1980. doi:10.1145/800141.804655.

4 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley, 2004.

5 John Augustine, William K. Moses, and Gopal Pandurangan. Brief Announcement: Distributed MST Computation in the Sleeping Model: Awake-Optimal Algorithms and Lower Bounds. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC, pages 51–53. ACM, 2022. doi:10.1145/3519270.3538459.

6 Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences*, 68(4):702–732, 2004. doi:10.1016/J.JCSS.2003.11.006.

7 Michael Ben-Or. Lower Bounds for Algebraic Computation Trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, STOC, pages 80–86. ACM, 1983.

8 Joshua Brody, Amit Chakrabarti, Ranganath Kondapally, David P. Woodruff, and Grigory Yaroslavtsev. Beyond set disjointness: the communication complexity of finding the intersection. In *ACM Symposium on Principles of Distributed Computing*, PODC, pages 106–113. ACM, 2014. doi:10.1145/2611462.2611501.

9 Harry Buhrman, Christoph Dürr, Mark Heiligman, Peter Høyer, Frédéric Magniez, Miklos Santha, and Ronald de Wolf. Quantum Algorithms for Element Distinctness. *SIAM Journal on Computing*, 34(6):1324–1330, 2005. doi:10.1137/S0097539702402780.

10 Wan Fokkink and Jun Pang. Simplifying Itai-Rodeh Leader Election for Anonymous Rings. *Electronic Notes in Theoretical Computer Science*, 128(6):53–68, 2005. doi:10.1016/J.ENTCS.2005.04.004.

11 Mohsen Ghaffari, Christoph Grunau, and Václav Rozhoň. Improved deterministic network decomposition. In *Proceedings of the 32nd Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 2904–2923. SIAM, 2021. doi:10.1137/1.9781611976465.173.

12 Dima Grigoriev, Marek Karpinski, Friedhelm Meyer auf der Heide, and Roman Smolensky. A Lower Bound for Randomized Algebraic Decision Trees. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, pages 612–619. ACM, 1996. doi:10.1145/237814.238011.

13 Johan Håstad and Avi Wigderson. The Randomized Communication Complexity of Set Disjointness. *Theory of Computing*, 3(1):211–219, 2007. doi:10.4086/TOC.2007.V003A011.

14 Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990. doi:10.1016/0890-5401(90)90004-2.

15 Ralph E. Johnson and Fred B. Schneider. Symmetry and Similarity in Distributed Systems. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing*, PODC, pages 13–22. ACM, 1985. doi:10.1145/323596.323598.

16 Bala Kalyanasundaram and Georg Schnitger. The Probabilistic Communication Complexity of Set Intersection. *SIAM Journal on Discrete Mathematics*, 5(4):545–557, 1992. doi:10.1137/0405044.

17 Hsiang-Tsung Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981. doi:10.1145/319566.319567.

18 Nathan Linial. Locality in Distributed Graph Algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

19 Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, SIGMOD, pages 677–689. ACM, 2015. doi:10.1145/2723372.2749436.

20 M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, 4 edition, 2020.

21 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.

22 Alexander A. Razborov. On the Distributional Complexity of Disjointness. *Theoretical Computer Science*, 106(2):385–390, 1992. doi:10.1016/0304-3975(92)90260-M.

23 Gerard Tel. Network orientation. *International Journal of Foundations of Computer Science*, 5(01):23–57, 1994. doi:10.1142/S0129054194000037.

24 Gerard Tel. *Introduction to distributed algorithms*. Cambridge university press, 2000.

25 Andrew Chi-Chih Yao. Some Complexity Questions Related to Distributive Computing (Preliminary Report). In *Proceedings of the 11th Annual ACM Symposium on Theory of Computing*, STOC, pages 209–213. ACM, 1979. doi:10.1145/800135.804414.

Appendix

A Pseudocode of the Deterministic Algorithms

Main algorithm executed at each node v.

```
$\texttt{build}_v \leftarrow true$, $\boldsymbol{x}^v \leftarrow v$'s input tokens, $\texttt{rid}_v \leftarrow \min\{\boldsymbol{x}^v\}$, $\texttt{p}_v \leftarrow \perp$, $\texttt{chi}_v \leftarrow \perp$, $\texttt{f}_v \leftarrow false$.
$\texttt{cnt}_v \leftarrow \perp$, $\texttt{ele}_v \leftarrow \top$, $\texttt{res}_v \leftarrow \perp$.        ▷ End of initialization.
for (each round) do
    for (each incident edge with label $i \in [\Delta_v]$) do
        $\texttt{ischild}_i \leftarrow \mathbb{I}[\texttt{p}_v == i]$. ▷ $\texttt{ischild}_i$ indicates whether edge $i$ connects to the parent of $v$.
        Send $\langle \texttt{res}_v, \texttt{build}_v, \texttt{rid}_v, \texttt{ischild}_i, \texttt{f}_v, \texttt{cnt}_v, \texttt{ele}_v \rangle$ through edge $i$.
    if ($\texttt{res}_v \neq \perp$) then Return $\texttt{res}_v$ as final result.        ▷ Termination.
    For $i \in [\Delta_v]$, let $m_i = \langle \texttt{res}_i, \texttt{build}_i, \texttt{rid}_i, \texttt{ischild}_i, \texttt{f}_i, \texttt{cnt}_i, \texttt{ele}_i \rangle$ be the message received via edge $i$.
    for (each edge $i \in [\Delta_v]$) do
        if ($\texttt{res}_i \neq \perp$) then $\texttt{res}_v \leftarrow \texttt{res}_i$.
        $\texttt{build}_v \leftarrow \texttt{build}_v \wedge \texttt{build}_i$.
    if ($\texttt{build}_v == true$) then Execute Procedure BFS-TREE-BUILDING.
    else Execute Procedure TOKEN-COLLISION-DETECTION.
```

Procedure BFS-TREE-BUILDING executed at node v.

```
$ID_v \leftarrow \{\texttt{rid}_i \mid i \in [\Delta_v]\}$.
if ($\min\{ID_v\} < \texttt{rid}_v$) then
    Let $j \in [\Delta_v]$ be one edge label satisfying $\texttt{rid}_j == \min\{ID_v\}$.
    $\texttt{rid}_v \leftarrow \texttt{rid}_j$, $\texttt{p}_v \leftarrow j$, $\texttt{f}_v \leftarrow false$.        ▷ Notice that $\texttt{f}_v$ is reset to $false$.
else if ($\max\{ID_v\} == \texttt{rid}_v$) then
    $\texttt{chi}_v \leftarrow \{i \mid i \in [\Delta_v]$ and $\texttt{ischild}_i == true\}$.
    if (($\forall i \in \texttt{chi}_v$, $\texttt{f}_i == true$) or $\texttt{chi}_v == \emptyset$) then $\texttt{f}_v \leftarrow true$.
if ($\texttt{p}_v == \perp$ and $\texttt{f}_v == true$) then $\texttt{build}_v \leftarrow false$.
```

Procedure TOKEN-COLLISION-DETECTION executed at node v.

```
$\texttt{chi}_v \leftarrow \{i \mid i \in [\Delta_v]$ and $\texttt{build}_i == false$ and $\texttt{ischild}_i == true$ and $\texttt{rid}_i == \texttt{rid}_v\}$.
Append $\{\texttt{ele}_i \mid i \in \texttt{chi}_v$ and $\texttt{ele}_i \in \{0,1\}^L\}$ to $\boldsymbol{x}^v$.
if ($\forall i \in [\Delta_v]$, $\texttt{build}_i == false$) then
    if (($\forall i \in \texttt{chi}_v$, $\texttt{cnt}_i \neq \perp$) or $\texttt{chi}_v == \emptyset$) then $\texttt{cnt}_v \leftarrow 1 + \sum_{i \in \texttt{chi}_v} \texttt{cnt}_i$.
    if ($\texttt{p}_v \neq \perp$) then
        if ($|\boldsymbol{x}^v| > 0$) then Eject one token from $\boldsymbol{x}^v$ and let that token be $\texttt{ele}_v$.
        else if (($\forall i \in \texttt{chi}_v$, $\texttt{ele}_i == \perp$) or $\texttt{chi}_v == \emptyset$) then $\texttt{ele}_v \leftarrow \perp$.
        else $\texttt{ele}_v \leftarrow \top$.
    else if ($\texttt{cnt}_v \neq \perp$ and (($\forall i \in \texttt{chi}_v$, $\texttt{ele}_i == \perp$) or $\texttt{chi}_v == \emptyset$)) then
        if (know value of $n$ and $\texttt{cnt}_v == n$ and no token collision in $\boldsymbol{x}^v$) then $\texttt{res}_v \leftarrow true$.
        else if (know value of $k$ and $|\boldsymbol{x}^v| == k$ and no token collision in $\boldsymbol{x}^v$) then $\texttt{res}_v \leftarrow true$.
        else $\texttt{res}_v \leftarrow false$.
```

Figure 1 Pseudocode of the deterministic token collision algorithm.

The complete pseudocode of the algorithm in Section 3 is given in Figure 1. Below are the explanations of some key variables that are used in the pseudocode. For any node v,

- $\texttt{build}_v$: a boolean variable indicating whether BFS-tree building is ongoing for v.
- $\texttt{rid}_v$: the identifier of v, intuitively it stores the root of the BFS-tree that v belongs to.
- $\texttt{p}_v$: the label of the edge connecting to the parent of v.
- $\texttt{chi}_v$: the set of edge labels representing the children of v.

- $\mathtt{f}_v$: a boolean variable indicating whether the BFS-tree rooted at v is fully constructed.
- $\mathtt{cnt}_v$: the size of the BFS-tree rooted at v.
- $\mathtt{ele}_v$: if $\mathtt{ele}_v \notin \{\perp, \top\}$, then it is the token that v intends to send to its parent in the next round; if $\mathtt{ele}_v = \top$, it indicates that there may exist a token in the BFS-tree rooted at v that has not been transferred to v's parent; if $\mathtt{ele}_v = \perp$, it indicates that all tokens in the BFS-tree rooted at v have already been transferred to v's parent.
- $\mathtt{res}_v$: the result of the token collision problem, that is, the algorithm's output at node v.

The pseudocode of the algorithm introduced in Section 5, which deals with the case that tokens are large, are given in Figure 2.

B Token Collision for the Randomized Scenario

For the sake of completeness, in this section, we briefly discuss the round complexity of the token collision problem when randomization is allowed.

Randomized upper bound. We first describe a randomized algorithm that solves token collision with probability at least $1 - 1/k$ within $O(D \cdot \frac{\log((\log k)/\log n)}{\log n} + k \cdot \frac{\log k}{\log n} + \frac{L}{\log n})$ rounds, hence proving the upper bound part of Theorem 6.

To begin with, we elect a leader among the nodes that have at least one token as input. Notice that there are at most k such nodes. Hence, by letting each such node v uniformly and independently sample $\mathtt{id}_v \in \{0,1\}^{c \log k}$ for some sufficiently large constant c, there is a unique node $\hat{v}$ that obtains the global minimum $\mathtt{id}$ with probability at least $1 - 1/k^{c-2}$. If we let each node continuously broadcast the minimum $\mathtt{id}$ that it ever received, a size-n BFS tree rooted at $\hat{v}$ would be constructed with probability at least $1 - 1/k^{c-2}$. Moreover, by using the pipelining approach we introduced in Section 5, this process takes at most $O(D \cdot \frac{\log((\log k)/\log n)}{\log n} + \frac{\log k}{\log n})$ rounds (see Lemma 30).

Once a size-n BFS tree is built, the root – which is also the leader – will collect all tokens to determine whether collisions exist. Notice that with randomization, we do not have to transfer each token in its entirety. In particular, we can leverage the following fact on the collision probability of random hash function to reduce the length of each token.

▶ **Fact 33** ([8]). *For any set $S \subseteq [2^L]$ of size $|S| = k$ and any $\beta \geq 0$, there exists a random hash function $h : [2^L] \to [q]$ with $q = O(k^{2+\beta})$ such that, with probability at least $1 - 1/k^\beta$, it holds that $h(x) \neq h(y)$ for all $x, y \in S$ with $x \neq y$. Moreover, h can be constructed using $O(L)$ random bits.*

Therefore, after BFS-tree construction, the leader can generate $O(L)$ random bits for constructing the random hash function, and broadcast these bits to all nodes in $O(D + \frac{L}{\log n})$ rounds. Then, each node uses the random hash function to reduce the length of its tokens to $O(\log k)$ bits. Finally, the k tokens each of length $O(\log k)$ is aggregated to the root in $O(D + k \cdot \frac{\log k}{\log n})$ rounds.

Clearly, the total runtime of the algorithm is

$$O\left(D \cdot \frac{\log((\log k)/\log n)}{\log n} + k \cdot \frac{\log k}{\log n} + \frac{L}{\log n}\right),$$

and it succeeds with probability at least $1 - 1/k$.

Randomized lower bound. The lower bound part of Theorem 6 can be obtained in the same manner as the proof of Theorem 32 by using the randomized lower bound of set-disjointness mentioned in Fact 7. We omit its proof to avoid redundancy.

Main algorithm executed at each node v for large tokens.

```
$\texttt{build}_v \leftarrow true$, $\boldsymbol{x}^v \leftarrow v$'s input tokens, $\texttt{rid}_v \leftarrow \min\{\boldsymbol{x}^v\}$, $\texttt{p}_v \leftarrow \perp$, $\texttt{chi}_v \leftarrow \perp$, $\texttt{f}_v \leftarrow false$.
Initialize four vectors $\{\texttt{rid}_i\}_{i\in[\Delta_v]}$, $\{\texttt{sent}_i\}_{i\in[\Delta_v]}$, $\{\texttt{ele}_i\}_{i\in\{0,\cdots,\Delta_v\}}$, and $\{\texttt{sente}_i\}_{i\in\{0,\cdots,\Delta_v\}}$.
Set $\texttt{ele}_0 \leftarrow \top$; for any $i \in [\Delta_v]$, set $\texttt{sent}_i \leftarrow 0$, $\texttt{rid}_i \leftarrow 2^L - 1$, $\texttt{sente}_i \leftarrow 0$.
$M \leftarrow \lceil L/B \rceil$, $P \leftarrow \lceil (\log \lceil L/B \rceil)/B \rceil$.                ▷ End of initialization.
for (each iteration containing $\Theta(\frac{\log(L/B)}{B})$ rounds) do
    $l \leftarrow \texttt{sent}_v + 1$, $r \leftarrow \min(\texttt{sent}_v + P, M)$.
    for (each incident edge with label $i \in [\Delta_v]$) do
        $\texttt{ischild}_i \leftarrow \mathbb{I}[\texttt{p}_v == i]$.
        Send $\langle \texttt{res}_v, \texttt{build}_v, \texttt{rid}_v[l,\cdots,r], \texttt{sent}_v, \texttt{ischild}_i, \texttt{f}_v, \texttt{cnt}_v, \texttt{ele}_v \rangle$ through edge $i$.
    $\texttt{sent}_v \leftarrow r$.
    if ($\texttt{res}_v \neq \perp$) then Return $\texttt{res}_v$ as final result.                ▷ Termination.
    Let $m_i = \langle \texttt{res}_i, \texttt{build}_i, \texttt{rid}'_i, \texttt{sent}'_i, \texttt{ischild}_i, \texttt{f}_i, \texttt{cnt}_i, \texttt{ele}'_i \rangle$ be the message received via edge $i \in [\Delta_v]$.
    for (each edge $i \in [\Delta_v]$) do
        $l \leftarrow \texttt{sent}'_i + 1$, $r \leftarrow \min(\texttt{sent}'_i + P, M)$, $\texttt{sent}_i \leftarrow r$, $\texttt{rid}_i[l,\cdots,r] \leftarrow \texttt{rid}'_i$, fill $\texttt{rid}_i[r+1,\cdots,M]$ with 1.
        if ($\texttt{res}_i \neq \perp$) then $\texttt{res}_v \leftarrow \texttt{res}_i$.
        $\texttt{build}_v \leftarrow \texttt{build}_v \wedge \texttt{build}_i$.
    if ($\texttt{build}_v == true$) then Execute Procedure BFS-TREE-BUILDING for large tokens.
    else Execute Procedure TOKEN-COLLISION-DETECTION for large tokens.
```

Procedure BFS-TREE-BUILDING executed at node v for large tokens.

```
$ID_v \leftarrow \{\texttt{rid}_i \mid i \in [\Delta_v]\}$.
if ($\min\{ID_v\} < \texttt{rid}_v$) then
    if ($\texttt{p}_v \neq \perp$ and $\texttt{rid}_{\texttt{p}_v} == \min\{ID_v\}$) then $j \leftarrow \texttt{p}_v$.
    else Let $j \in [\Delta_v]$ be one edge label satisfying $\texttt{rid}_j == \min\{ID_v\}$.
    $\texttt{rid}_v \leftarrow \texttt{rid}_j$, $\texttt{sent}_v \leftarrow \texttt{sent}'_j$, $\texttt{p}_v \leftarrow j$, $\texttt{f}_v \leftarrow false$.        ▷ Notice that $\texttt{f}_v$ is reset to $false$.
else if ($\max\{ID_v\} == \texttt{rid}_v$ and $\texttt{sent}_v == M$ and $(\forall i \in [\Delta_v], \texttt{sent}_i == M)$) then
    $\texttt{chi}_v \leftarrow \{i \mid i \in [\Delta_v]$ and $\texttt{ischild}_i == true\}$.
    if ($(\forall i \in \texttt{chi}_v, \texttt{f}_i == true)$ or $\texttt{chi}_v == \emptyset$) then $\texttt{f}_v \leftarrow true$.
if ($\texttt{p}_v == \perp$ and $\texttt{f}_v == true$) then $\texttt{build}_v \leftarrow false$.
```

Procedure TOKEN-COLLISION-DETECTION executed at node v for large tokens.

```
$\texttt{chi}_v \leftarrow \{i \mid i \in [\Delta_v]$ and $\texttt{build}_i == false$ and $\texttt{ischild}_i == true$ and $\texttt{sent}_i == M$ and $\texttt{rid}_i == \texttt{rid}_v\}$.
for ($i \in \texttt{chi}_v$ and $\texttt{ele}'_i \notin \{\top, \perp\}$) do
    $\texttt{ele}_i[\texttt{sente}_i + 1, \cdots, \min(\texttt{sente}_i + P, M)] \leftarrow \texttt{ele}'_i$, $\texttt{sente}_i \leftarrow \min(\texttt{sente}_i + P, M)$.
    if ($\texttt{sente}_i == M$) then Append $\texttt{ele}_i$ to $\boldsymbol{x}^v$ and set $\texttt{sente}_i \leftarrow 0$.
if ($(\forall i \in [\Delta_v], \texttt{build}_i == false)$ and $\texttt{sent}_v == M$) then
    if ($(\forall i \in \texttt{chi}_v, \texttt{cnt}_i \neq \perp)$ or $\texttt{chi}_v == \emptyset$) then $\texttt{cnt}_v \leftarrow 1 + \sum_{i\in\texttt{chi}_v} \texttt{cnt}_i$.
    if ($\texttt{p}_v \neq \perp$) then
        if ($\texttt{sente}_0 \neq 0$) then
            $l \leftarrow \texttt{sente}_0 + 1$, $r \leftarrow \min(\texttt{sente}_0 + P, M)$, $\texttt{ele}_v \leftarrow \texttt{ele}_0[l,\cdots,r]$, $\texttt{sente}_0 \leftarrow r \bmod M$.
        else if ($|\boldsymbol{x}^v| > 0$) then
            Eject one token from $\boldsymbol{x}^v$ and let that token be $\texttt{ele}_0$.
            $\texttt{ele}_v \leftarrow \texttt{ele}_0[1,\cdots,P]$, $\texttt{sente}_0 \leftarrow P$.
        else if ($(\forall i \in \texttt{chi}_v, \texttt{ele}_i == \perp)$ or $\texttt{chi}_v == \emptyset$) then $\texttt{ele}_v \leftarrow \perp$.
        else $\texttt{ele}_v \leftarrow \top$.
    else if ($\texttt{cnt}_v \neq \perp$ and $((\forall i \in \texttt{chi}_v, \texttt{ele}_i == \perp)$ or $\texttt{chi}_v == \emptyset)$) then
        if (know value of $n$ and $\texttt{cnt}_v == n$ and no token collision in $\boldsymbol{x}^v$) then $\texttt{res}_v \leftarrow true$.
        else if (know value of $k$ and $|\boldsymbol{x}^v| == k$ and no token collision in $\boldsymbol{x}^v$) then $\texttt{res}_v \leftarrow true$.
        else $\texttt{res}_v \leftarrow false$.
```

Figure 2 Pseudocode of the deterministic token collision algorithm for large tokens.

Asynchronous Fault-Tolerant Distributed Proper Coloring of Graphs

Alkida Balliu
Gran Sasso Science Institute, L'Aquila, Italy

Pierre Fraigniaud
IRIF - CNRS & Univ. Paris Cité, France

Patrick Lambein-Monette
Unaffiliated

Dennis Olivetti
Gran Sasso Science Institute, L'Aquila, Italy

Mikaël Rabie
IRIF - Université Paris Cité, France

Abstract

We revisit *asynchronous* computing in networks of *crash-prone* processes, under the asynchronous variant of the standard LOCAL model, recently introduced by Fraigniaud et al. [DISC 2022]. We focus on the vertex coloring problem, and our contributions concern both lower and upper bounds for this problem.

On the upper bound side, we design an algorithm tolerating an arbitrarily large number of crash failures that computes an $O(\Delta^2)$-coloring of any n-node graph of maximum degree Δ, in $O(\log^\star n)$ rounds. This extends Linial's seminal result from the (synchronous failure-free) LOCAL model to its asynchronous crash-prone variant. Then, by allowing a dependency on Δ on the runtime, we show that we can reduce the colors to $\left(\frac{1}{2}(\Delta+1)(\Delta+2)-1\right)$. For cycles (i.e., for $\Delta = 2$), our algorithm achieves a 5-coloring of any n-node cycle, in $O(\log^\star n)$ rounds. This improves the known 6-coloring algorithm by Fraigniaud et al., and fixes a bug in their algorithm, which was erroneously claimed to produce a 5-coloring.

On the lower bound side, we show that, for $k < 5$, and for every prime integer n, no algorithm can k-color the n-node cycle in the asynchronous crash-prone variant of LOCAL, independently from the round-complexities of the algorithms. This lower bound is obtained by reduction from an original extension of the impossibility of solving *weak symmetry-breaking* in the wait-free shared-memory model. We show that this impossibility still holds even if the processes are provided with inputs susceptible to help breaking symmetry.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases LOCAL model, Graph Coloring, Renaming, Weak Symmetry-Breaking, Fault-Tolerance, Wait-Free Computing

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.5

Related Version *Full Version*: `https://arxiv.org/abs/2408.10971`

Funding Partially funded by MUR (Italy) Department of Excellence 2023 - 2027, the PNRR MIUR research project GAMING "Graph Algorithms and MinINg for Green agents" (PE0000013, CUP D13C24000430001), and by the French ANR projects DUCAT (ANR-20-CE48-0006) and QuDATA (ANR-18-CE47-0010).

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 5; pp. 5:1–5:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

1.1 Asynchrony, Failures, and Networks

To what extent a global solution to a computational problem can be obtained from locally available data? What can be computed locally? These are some of the questions that were asked, and partially answered 30 years ago in two seminal papers [23, 25] in the field of distributed network computing. Since then, tremendous progress has been made about these questions, and even detailed books [22, 27] can only touch a small fraction of the content of the current literature on this topic. Nevertheless, the vast majority of the achievements on *local computing* have been obtained in *synchronous failure-free* models, among which the most common ones are referred to as LOCAL [23] and CONGEST [27].

In both models, processing nodes occupy the vertices of a graph, and exchange messages along the edges of that graph. They all start at the same time, and computing proceeds as a sequence of synchronous rounds. At each round, every pair of adjacent nodes can exchange messages (one in each direction), and every node can perform some individual computation. CONGEST differs from LOCAL only as far as the message size is concerned: messages are bounded to be of size at most B bits in CONGEST (it is common to set $B = O(\log n)$). There are at least two solid reasons why such elegant but simplistic models should be considered. First, they ideally capture the notion of *spatial locality*, as algorithms performing in t rounds produce an output at each node that is solely based on the t-neighborhood of the node. Second, the existence of efficient synchronizers [3, 4, 19] enables to implement algorithms designed for synchronous models on asynchronous networks, with only limited slowdown.

Yet, models such as LOCAL and CONGEST suffer from one notable limitation: they ignore the potential presence of failures. Indeed, transient failures have been addressed in the framework of *self-stabilization*, but crash or malign failures are mostly ignored in the framework of local computing in networks. Instead, studying the interplay of asynchrony and failures has been the main topic of interest of distributed computing in general [2, 24, 28], since the seminal “FLP impossibility result” stating that consensus is impossible in asynchronous systems with failures, even under the restriction that at most one crash failure may occur [15]. However, the design of algorithms dedicated to asynchronous crash-prone systems have been mostly performed in *shared-memory* or *message-passing* models: the former assumes that processes exchange information by writing and reading in a shared memory; the latter assumes that any two processes can exchange messages directly along a private channel. While these two models are excellent abstractions of very many types of distributed systems, ranging from multi-core architectures to large-scale computing platforms, they do not enable the study of spatial locality, as the structure of the physical network is abstracted away.

An attempt to resolve this tension between synchronous failure-free computing in networks, and asynchronous computing in crash-prone systems has been recently proposed [16], by considering asynchronous networks subject to crash failures.

1.2 The ASYNC LOCAL Model

The asynchronous crash-prone LOCAL model[1] (ASYNC LOCAL in short), introduced in [16], aims at capturing a setting that is a hybrid between shared memory and network computing.

[1] One could also consider the variant ASYNC CONGEST of ASYNC LOCAL by limiting to $O(\log n)$ bits the size of the registers in which nodes read and write, but we restrict ourselves to the LOCAL variant, as standard wait-free computing does not generally restrict the size of the registers.

This model can be described conceptually in two possible ways (see Section 2.1 for more details):

- The ASYNC LOCAL model can be viewed as the standard wait-free shared-memory model [2, 21] in which the read-access to other process's registers is restricted. It bears similarities with the *atomic state* model in self-stabilization [9]. In an n-process system, each process $i \in [n]$ can solely read the registers of processes $j \in N_G(i)$, where $N_G(i)$ denotes the set of neighbors of vertex i in a graph G. That is, the wait-free shared-memory model is the ASYNC LOCAL model in which the graph G is fixed to be the complete graph (or clique) K_n.
- The ASYNC LOCAL model can alternatively be viewed as the standard LOCAL model [22, 27] in which each node writes in its local register(s) instead of sending messages, and reads the registers of its neighbors instead of receiving messages from them. In addition, ASYNC LOCAL allows asynchronous executions, that is, each process reads and writes at its own pace, which may vary with time, and it may even crash (i.e., stop functioning, and never recover). Note that, as for LOCAL, the graph G is unknown to the nodes in ASYNC LOCAL, as it is typically the *input* to the problems of interest in network computing.

In the framework of asynchronous computing, the computing elements are referred to as *processes*, whereas they are referred to as *nodes* in the context of computing in networks, but we use these two terms indistinctly. The terminology "wait-free" refers to the fact that (1) an arbitrarily large number of processes can crash, and (2) a node cannot distinguish whether a neighboring node has crashed or is simply slow, from which it follows that a node must never "wait" for some action performed by another node, and must terminate independently from which of the other nodes have crashed (unless itself has crashed).

It was shown in [16] that the computing power of ASYNC LOCAL is radically different from the one of LOCAL. Indeed, the authors proved that constructing a maximal independent set (MIS) is simply *impossible* in ASYNC LOCAL, even in the n-node cycles C_n, $n \geq 3$, while, on cycles, it just takes $\Theta(\log^\star n)$ rounds in LOCAL [12, 23]. However, the authors show also that proper coloring C_n is possible in ASYNC LOCAL, to the expense of using a larger palette of colors, i.e., 6 colors instead of just 3 as in LOCAL (a 5-coloring algorithm is also claimed in [16], but, as we shall show later, there is a bug in that algorithm). Indeed, a simple reduction to *renaming* (see [2] for the definition) shows that, under the ASYNC LOCAL model, no algorithms can proper color all graphs of maximum degree Δ using less than $2\Delta + 1$ colors whenever $\Delta + 1$ is a power of a prime. This is because ASYNC LOCAL and standard shared-memory coincides when the graph is a clique of $n = \Delta + 1$ nodes. The main result in [16] is a distributed asynchronous algorithm in the ASYNC LOCAL model that achieves proper 6-coloring of any n-node cycle, $n \geq 3$, in $O(\log^\star n)$ rounds, which is optimal thanks to [23]. In ASYNC LOCAL, the *round-complexity* of an algorithm is the maximum, taken over all nodes, and all executions, of the number of times a node writes in its register, and reads the registers of its neighbors.

1.3 Our results

In a nutshell, we show that there exists an algorithm for proper coloring graphs with maximum degree Δ in the ASYNC LOCAL model, using a palette of $\frac{1}{2}(\Delta+1)(\Delta+2)-1$ colors, resulting into a 5-coloring algorithm for the cycles. This result was obtained by first showing how to implement Linial's coloring algorithm in the asynchronous setting, and then by developing a new technique based on reallocating identifiers to nodes. Note that even implementing

Linial's coloring algorithm asynchronously is not straightforward, as it is not even clear whether the trivial recoloring algorithm that proceeds iteratively over all color classes can be implemented in the ASYNC LOCAL model. Moreover, we show that, for infinitely many values of n, 5-coloring the n-node cycles is the best that can be achieved in ASYNC LOCAL. This significantly improves the lower bound in [16] on the number of colors required for proper coloring cycles under ASYNC LOCAL, which held for $n = 3$ only.

Obtaining our lower bound required to revisit entirely the known lower bound on weak symmetry breaking[2] in the standard asynchronous shared-memory model, by considering the impact of a priori "knowledge" given to the processes. For instance, if the processes know a priori that one process is given advice 0, and one process is given advice 1, then weak symmetry breaking becomes trivially solvable. For which a priori knowledge weak symmetry breaking becomes trivially solvable, and for which it remains unsolvable? We show that answering this novel question for specific types of a priori knowledge results into new impossibility results for the standard asynchronous shared-memory model, which translate into lower bounds and impossibility results in the ASYNC LOCAL model.

We stress the fact that while all (Turing computable) tasks are solvable in the LOCAL model, not all taks are solvable in ASYNC LOCAL, yet we also address *complexity* issues, by showing that, for constant Δ, our $\big(\frac{1}{2}(\Delta+1)(\Delta+2)-1\big)$-coloring algorithm performs in $O(\log^\star n)$ rounds in ASYNC LOCAL, that is, as fast as the $\Omega(\log^\star n)$ lower bound [23] on the number of rounds required for coloring cycles in the synchronous failure-free LOCAL model. These results are detailed next.

1.3.1 Proper Coloring

We mostly focus on distributed proper coloring, arguably one of the most important and thoroughly studied symmetry-breaking tasks in network computing – see, e.g., [17, 18, 20] for recent results on the matter[3]. First, we show that Linial's technique from [23] based on cover-free families of set systems can be used asynchronously, for the design of an $O(\Delta^2)$-coloring of graphs of maximum degree Δ, running in $O(\log^\star n)$ rounds in n-node graphs under ASYNC LOCAL. Then we show that the approach from [16] for 6-coloring cycles can be generalized to color arbitrary graphs. Specifically, we design an algorithm computing a $\frac{(\Delta+1)(\Delta+2)}{2}$-coloring in graphs of maximum degree Δ running in $O(\log^\star n) + f(\Delta)$ rounds under ASYNC LOCAL, where the additional term $f(\Delta)$ depends on Δ only. This line of results culminates in the design of an algorithm enabling to save one color, i.e., that computes a $\big(\frac{(\Delta+1)(\Delta+2)}{2}-1\big)$-coloring, still running in $O(\log^\star n) + f(\Delta)$ for some function f. Reducing the color palette by just one color may seem of little importance, but it is not, for two reasons. First, a palette of size $\frac{(\Delta+1)(\Delta+2)}{2} - 1$ is the best that we are aware of for which it is possible to proper color all graphs of maximum degree Δ in $O(\log^\star n)$ rounds in ASYNC LOCAL (ignoring the additional term depending on Δ only). Saving one more color appears to be challenging. Second, in the case of cycles, i.e., $\Delta = 2$, this allows us to fix a bug in the 5-coloring algorithm from [16]. Indeed, this latter algorithm is shown to be erroneous, as

[2] Weak symmetry breaking is the task in which processes start with no inputs, and each process must output 0 or 1, under the constraint that, whenever all processes terminate, at least one process must output 0, and at least one process must output 1.

[3] In the context of distributed computing in networks, especially in the LOCAL and CONGEST models, one is interested in properly coloring graphs with maximum degree Δ using a palette of $f(\Delta)$ colors, where $f(\Delta)$ grows slowly with Δ. One typical example is $f(\Delta) = \Delta + 1$ as all graphs of maximum degree Δ can be properly colored with $\Delta + 1$ colors, but one is also interested in larger functions f, e.g., $f(\Delta) = \Theta(\Delta^2)$, whenever this choice enables to obtain faster algorithms.

there are schedulings of the nodes that result in livelocks preventing the algorithm from terminating. Nevertheless, our algorithm shows that 5-coloring the n-node cycles in $O(\log^\star n)$ rounds under ASYNC LOCAL is indeed possible.

1.3.2 Lower Bounds and Impossibility Results

Our second line of contribution is related to lower bounds on the size of the color palette enabling to proper color graphs asynchronously. It was observed in [16] that since the class of graphs with maximum degree Δ includes the clique with $n = \Delta + 1$ nodes, and since *renaming* [2] in a set of less than $2N - 1$ names cannot be done wait-free in N-process shared-memory systems whenever N is a power of a prime, proper coloring graphs of maximum degree Δ in ASYNC LOCAL cannot be achieved with a color palette smaller than $2\Delta + 1$ colors, i.e., 5 colors in the case of cycles (independently from the number of rounds). However, the question of whether one can 4- or even 3-color long cycles (i.e., excluding the specific case of the clique C_3) under ASYNC LOCAL was left open in [16]. We show that this is impossible whenever n is prime, that is, there are infinitely many values of n for which 5-coloring the n-node cycle is the best that can be achieved in ASYNC LOCAL.

1.3.3 Reduction from Weak Symmetry-Breaking with Inputs

We achieve our lower bound on the number of colors thanks to a result of independent interest in the standard framework of wait-free shared-memory computing. We show that there are no symmetric wait-free algorithms solving *weak symmetry-breaking* [2] in n-process asynchronous shared-memory systems whenever n is prime, *even if processes are provided with inputs from a non-prime-divisible and order-invariant set of inputs.* We achieve this impossibility result by extending the proof in [1] for weak symmetry-breaking to the case in which processes have inputs that do not trivially break symmetry. Our impossibility result for weak symmetry-breaking with inputs has other consequences on the ASYNC LOCAL model, including the facts that weak 2-coloring is impossible in cycles of prime size, and that, for every even $\Delta \geq 2$, there is an infinite family of regular graphs for which $(\Delta + 2)$-coloring cannot be solved in ASYNC LOCAL.

Finally, using different techniques, we also show that even a weak variant of maximal independent set (MIS) cannot be solved in cycles with at least 7 nodes, and that, for every $\Delta \geq 2$, $(\Delta + 1)$-coloring trees of maximum degree Δ is impossible under ASYNC LOCAL.

1.4 Related Work

The combination of asynchrony *and* failures in the general framework of distributed computing in networks has been studied a lot in the context of *self-stabilization.* The latter deals with *transient* failures susceptible to modify the content of some of the variables defining the states of the nodes. The role of a self-stabilizing algorithm is therefore to guarantee that if the network is in an illegal configuration (i.e., a configuration not satisfying some specific correctness condition), then it will automatically return to a legal configuration, and will remain in a legal configuration, unless some other failure(s) occur. Self-stabilizing graph coloring algorithms have been designed [5, 6, 7, 8]. However, these algorithms provide solutions only for executions during which there are no failures. Instead, in ASYNC LOCAL, failures may occur at any time during the execution, and once a process crashes it never recovers. This has important consequences on what can or cannot be computed in ASYNC LOCAL. For instance, 3-coloring the n-node cycle is possible in a self-stabilizing manner for every $n \geq 3$, while we show that even 4-coloring the n-node cycle is impossible for infinitely many n (namely, for all prime n).

It is also worth mentioning [11, 13], which introduced the DECOUPLED model, where crash-prone processes occupy the nodes of a *reliable* and *synchronous* network. The DECOUPLED model is stronger than ASYNC LOCAL, and indeed it was shown that if there exists an algorithm solving a task in the LOCAL model, then there exists an algorithm solving that task in the DECOUPLED model as well, with limited slowdown. Instead, we show that even a weak variant of MIS is impossible in large cycles under ASYNC LOCAL.

Another field of research very much related to our work is the study of *synchronous* networks with failures, whether it be crash or even malicious process failures, or message omission failures (see, e.g., [10, 26, 29, 30]). In these models, the focus has mostly been put on the study of tasks such as consensus and set-agreement. The ASYNC LOCAL model somehow mixes some of the key aspects of the models considered in these work, including the presence of crash failures, and the fact that the communications are mediated by a graph distinct from the complete graph. The same way standard wait-free computing in shared-memory systems can be viewed as one specific instance of the oblivious message adversary model, wait-free computing in the ASYNC LOCAL model in a graph G may be viewed as the instance of the oblivious message adversary model in which messages can only be sent along the edges of the graph G. We however focus on solving graphs problems such as coloring or independent set, motivated by the need to solve various symmetry breaking problems in networks, including frequency assignment and cluster decomposition. For such problems, it is more more convenient to use the framework of ASYNC LOCAL, in which the graph G is part of the input, as in the LOCAL model.

2 Model and Definitions

We first recall the ASYNC LOCAL model as introduced in [16], and then provide an example for an algorithm in this model.

2.1 The ASYNC LOCAL model

Like the LOCAL model [27], the ASYNC LOCAL model assumes a set of $n \geq 1$ processes, each process occupying a distinct node of an n-vertex graph $G = (V, E)$, which is supposed to be simple and connected. Each process, i.e., each node $v \in V$, has an identifier id_v that is supposed to be unique in the graph. The identifiers are not necessarily between 1 and n, but they are supposed to be stored on $O(\log n)$ bits. That is, all node identifiers lie in the integer interval $[1, N]$ for some bound $N = \mathrm{poly}(n)$. Like in the asynchronous shared-memory model, every node v comes equipped with a single-writer/multiple-reader register $R(v)$ in which it can write values. However, in contrast with the shared-memory model, *only v's neighbors in the graph G* are able to read its register $R(v)$, and symmetrically, node v can only read the registers $R(w)$ of nodes $w \in N_G(v) = \{u \in V \mid \{u, v\} \in E\}$. We assume that each node can write in its register, and then read all its neighbors' registers, in a single atomic operation. Neighboring nodes can perform this write&read operation concurrently, in which case they both read the value concurrently written in the other node's register. This communication primitive is thus akin to an *immediate snapshot* object with read accesses mediated by a graph, in a similar manner to the *atomic state* model in the context of self-stabilizing algorithms [9]. Computation proceeds asynchronously, and each node may crash, in which case it stops functioning, and it never recovers. Therefore, in the particular case of the clique $G = K_n$, ASYNC LOCAL boils down to the standard asynchronous crash-prone shared-memory model with immediate snapshots [2]. The registers are of unbounded size. Therefore, as in the LOCAL model, and as in most wait-free computing models [21] as well, we can assume *full-information protocols*, in which every node writes its entire state in its register, and read the states of its neighbors in their registers.

Remark. Due to its nature, the ASYNC LOCAL model may have also been named "iterated immediate local snapshot". Nevertheless, for its close connection to the standard LOCAL model used for the study of graph problems (e.g., coloring) in distributed computing, we preferred to stick to the terminology ASYNC LOCAL.

Input. In addition to its identifier id_v, every node v may be provided with some input, denoted by input_v. The latter may be the number n of nodes in the graph, or an upper bound N on n, or any label $\ell(v) \in \{0,1\}^*$ whose semantic depends on the context (e.g., it may represent a boolean mark, or a color, etc.). Note that the network G is typically unknown to the nodes, even if some specific parameters may be provided to each node as input, such as the maximum degree Δ of G.

Algorithm. An algorithm $\mathcal{A}$ for the ASYNC LOCAL model may be described by two functions:

- Init: used to initialize the state of each node, as a function of its input;
- Alg: used to update the state of a node, as a function of its current state, and of the states of its neighbors.

Scheduling. An execution of an algorithm $\mathcal{A}$ depends on how the nodes are scheduled. A *scheduling* is a sequence $\mathcal{S} = S_1, S_2, \dots$ of subsets $S_i \subseteq V$ of nodes. For every $i \geq 1$, the set S_i denotes the set of nodes that are activated at *step* i. Each of these nodes performs an immediate-snapshot, and updates its state accordingly. For instance, the scheduling $\{u,v\}, \{v\}, \{v\}, \{v\}, \dots$ represents the execution in which nodes u and v run concurrently at the first step, and then v runs solo, i.e., v is the only node activated at every step $i \geq 2$. That is, u has crashed after step 1, and all the nodes $w \notin \{u,v\}$ had crashed initially, none of them taking any step. Instead, the scheduling $V, V, V, \dots$ represents a synchronous execution in which no node crashes.

Full-Information Protocols. For every $v \in V$, let $\mathrm{OldState}_{v,1} \leftarrow \bot$, and $\mathrm{NewState}_{v,1} \leftarrow \mathrm{Init}(\mathrm{id}_v, \mathrm{input}_v)$. For every $i \geq 1$, the variable $\mathrm{OldState}_{v,i}$ represents what a neighbor of v gets whenever reading the memory of v, and $\mathrm{NewState}_{v,i}$ represents the updated state of v, which will become visible to its neighbors the next time v is scheduled. More specifically, for every $i \geq 1$, if $v \notin S_i$, then $\mathrm{OldState}_{v,i+1} \leftarrow \mathrm{OldState}_{v,i}$ and $\mathrm{NewState}_{v,i+1} \leftarrow \mathrm{NewState}_{v,i}$. Instead, if $v \in S_i$, then $\mathrm{OldState}_{v,i+1} \leftarrow \mathrm{NewState}_{v,i}$, and $\mathrm{NewState}_{v,i+1} \leftarrow \mathrm{Alg}(\mathrm{OldState}_{v,i+1}, \{\mathrm{OldState}_{u,i+1} \mid u \in N_G(v)\})$. In other words, all nodes that are scheduled at step i write their current state, then read the state of their neighbors, and then use the obtained knowledge in order to update their state. The new states resulting from these updates will become visible to their neighbors the next time that they are scheduled. That is, we model a setting in which writing and then reading the state of the neighbors is an atomic operation, but it may take some time to compute a new state.

Termination. We let $\mathrm{Terminated}(x)$ be a special state denoting that a node terminates with output x. If a node v satisfies $\mathrm{NewState}_{v,t} = \mathrm{Terminated}(x)$ at some step $t \geq 1$, then v decides the output x, and it is assumed that if v is scheduled again in the future, then its state does not change, that is, $\mathrm{NewState}_{v,t+i} = \mathrm{NewState}_{v,t}$ for all $i \geq 1$.

Algorithm 1 An algorithm for 6-coloring cycles. Code of node v, with sole input id_v.

```
procedure CycleSixColoring($\mathrm{id}_v$)
    $x \leftarrow \mathrm{id}_v$;   $a \leftarrow 0$;   $b \leftarrow 0$;            ▷ $(x, a, b)$ is the state $s$ of $v$
    repeat forever
        $(s_1, s_2) \leftarrow \mathrm{WriteSnapshot}(s)$   ▷ $s_1$ and $s_2$ are the states of the two neighbors of $v$
        if $(a, b) \notin \{(s_1.a, s_1.b), (s_2.a, s_2.b)\}$ then return $(a, b)$
        else        ▷ In the following: $s_i = \bot \implies (s_i.x = \bot) \wedge (s_i.a = \bot) \wedge (s_i.b = \bot)$.
            $a \leftarrow \min \mathbb{N} \smallsetminus \{s_i.a \mid (i \in \{1,2\}) \wedge (s_i \neq \bot) \wedge (s_i.x > x)\}$
            $b \leftarrow \min \mathbb{N} \smallsetminus \{s_i.b \mid (i \in \{1,2\}) \wedge (s_i \neq \bot) \wedge (s_i.x < x)\}$
        end if
    end repeat
end procedure
```

Round complexity. The runtime of a node v is defined as

$$T_v = |\{i \geq 1 \mid v \in S_i \text{ and } \mathrm{NewState}_{v,i} \neq \mathrm{Terminated}(x) \text{ for any possible output } x\}|.$$

That is, the runtime of a v is equal to how many times v is scheduled before it terminates. The runtime of an algorithm on a graph $G = (V, E)$ is then $\max\{T_v \mid v \in V\}$. The runtime of an algorithm in a graph class $\mathcal{G}$ is the maximum runtime of the algorithm, over all graphs $G \in \mathcal{G}$. The runtime of an algorithm may depend on the identifiers given to the nodes. However, as said before, we use the standard assumption that the identifiers are from the interval $[1, N]$ where $N = \mathrm{poly}(n)$. The runtime is thus typically expressed as a function of n (the order of the graph) and Δ (the maximum degree of the graph). The complexity of a problem is the minimum runtime (as a function of n and Δ) among all possible algorithms that solve the problem. The typical graph class we are interested in is $\mathcal{G}_\Delta$, the class of all graphs with maximum degree Δ.

Remark. In absence of failures, and if all nodes run synchronously, the runtime of an algorithm in the ASYNC LOCAL model is identical to its runtime in the LOCAL model.

2.2 Algorithm Description

While an algorithm can be formally described by providing the two functions Init and Alg, we now describe an alternative, and possibly easier way of describing an algorithm. An example is provided in Algorithm 1 from [16], which is aiming at solving 6-coloring in cycles. This algorithm uses the function WriteSnapshot(s), which allows to perform an immediate snapshot (i.e., a write of the current state s immediately followed by a snapshot of all the states of the neighbors), and uses the function return, which explicitly provides the output (instead of using Terminated(x)).

In Algorithm 1, the state s of each (non terminated) node is a triplet $s = (x, a, b)$ of natural numbers. Given a state s, $s.x$, $s.a$, and $s.b$ respectively denote the first, second, and third element in s. The state of a terminated node is a pair (a, b) of natural numbers. One can check (see [16]) that the output pairs (a, b) can take at most 6 different values.

The state s of a node v is updated by updating some of all of its components x, a, or b. Actually, the entry $x = \mathrm{id}_v$ does not change. The entry a is updated to the smallest natural number excluding the a-values used by neighbors of larger identifiers, and b is updated to the smallest natural number excluding the b-values used by the neighbors of smaller identifiers. These values are equal to $\bot$ if they have not yet been written in the register (i.e., if a neighbor has not yet performed a single write). If a node v notices that its current state (x, a, b) is such that (a, b) is different from the (a, b)-pairs of both neighbors, then v terminates, and decides color (a, b). An example of an execution of Algorithm 1 is provided in Appendix A.

3 Results and Road Map

We have now all ingredients sufficient to formally state our results.

3.1 Algorithms for ASYNC LOCAL

We first show (cf. Section 4) that Linial's $O(\Delta^2)$-coloring algorithm can be adapted to work in the asynchronous wait-free setting.

▶ **Theorem 1.** *For every $\Delta \geq 2$, the round-complexity of $O(\Delta^2)$-coloring graphs of maximum degree Δ in the ASYNC LOCAL model is $O(\log^* n)$.*

Then, we show (cf. Section 5) that, at the cost of increasing the runtime by an additive factor depending on Δ, it is possible to reduce the number of colors from $O(\Delta^2)$ to $(\Delta + 1)(\Delta + 2)/2$.

▶ **Theorem 2.** *For every $\Delta \geq 2$, the round-complexity of $\frac{1}{2}(\Delta + 1)(\Delta + 2)$-coloring graphs of maximum degree Δ in the ASYNC LOCAL model is $O(\log^* n) + f(\Delta)$, where f is a function depending on Δ only.*

Finally, we show (cf. Section 6) that we can exploit the fact that the coloring produced by Theorem 2 satisfies special properties for reducing the size of the color palette by one.

▶ **Theorem 3.** *For every $\Delta \geq 2$, the round-complexity of $(\frac{1}{2}(\Delta + 1)(\Delta + 2) - 1)$-coloring graphs of maximum degree Δ in the ASYNC LOCAL model is $O(\log^* n) + f(\Delta)$, for some function f that only depends on Δ.*

An important consequence of this result is the case $\Delta = 2$. Theorem 3 shows that there is an algorithm for 5-coloring cycles. While such an algorithm was already claimed to exist in [16], we show (cf. Appendix B) that the algorithm supporting that claim is erroneous. Specifically, we provide an instance in which the algorithm does not terminate. Theorem 3 provides a novel algorithm, which allows us to establish the following result.

▶ **Corollary 4.** *The round-complexity of 5-coloring cycles in the ASYNC LOCAL model is $O(\log^* n)$.*

3.2 Impossibility Results

As pointed out in [16] several impossibility results for ASYNC LOCAL are mere consequences of the fact the this model coincides with the standard wait-free shared-memory model whenever the underlying graph G is a clique K_n. This is for instance the case of the impossibility of 4-coloring C_3 (by reduction from renaming), and the impossibility of constructing a maximal independent set, i.e., MIS (by reduction from strong symmetry breaking). Whether or not it is possible to 4-color cycles C_n for $n > 3$ was left open in [16]. We show that, for infinitely many values of n, the problem of 4-coloring the n-node cycle C_n is not solvable in ASYNC LOCAL. To establish this result, we prove a result of independent interest, in the framework of wait-free shared memory computing. Specifically, we extend the proof in [1] that weak symmetry breaking is impossible in the wait-free shared memory systems. We show that this problem remains impossible even if some input are provided to the processes, which may potentially help them to break symmetry. The set of possible inputs has to agree with some restrictions, called non-prime-divisible and order-invariant (with respect to a particular subset of processes). Roughly, the set of possible input assignments must not be divisible by the number n of processes whenever n is prime, and it must be closed under permuting the identifiers of a particular subset of the processes by an order-invariant permutation. Also recall that an algorithm is *symmetric* if for every execution α on a subset P of processes, and for every permutation $\pi : [n] \to [n]$ order preserving on P, we have that, for every $i \in P$, process i outputs x in α if and only if process $\pi(i)$ outputs x on the execution $\pi(\alpha)$ resulting from permuting the scheduling of the processes in P according to α. Our impossibility results are shown in the full version.

▶ **Theorem 5.** *Let n be a prime number. There are no symmetric wait-free deterministic algorithms solving weak symmetry break in the asynchronous wait-free shared memory model with n processes, even if the processes are provided with inputs from a non-prime-divisible and order-invariant set of inputs.*

Theorem 5 has three important consequences.

▶ **Corollary 6.** *Let $n \geq 3$ be a prime number. The problem of 4-coloring the n-node cycle cannot be solved deterministically in ASYNC LOCAL.*

A weaker form of symmetry breaking is weak 2-coloring [25]. It is required to 2-color the input graph such that every (non isolated) node has at least one neighbor colored with a different color.

▶ **Corollary 7.** *Let $n \geq 3$ be a prime number. The problem of weak 2-coloring the n-node cycle cannot be solved deterministically in ASYNC LOCAL.*

Finally, we prove that, for even values of Δ, there are a infinitely many Δ-regular graphs that cannot be $(\Delta + 2)$-colored in ASYNC LOCAL. This extends the lower bound of $2\Delta + 1$ colors, which applies only for the clique of $\Delta + 1$ nodes with $\Delta + 1$ power of a prime, to an infinite family of graphs with maximum degree Δ.

▶ **Corollary 8.** *Let Δ be an even number, and let $n > \Delta$ be a prime number. The problem of $(\Delta+2)$-coloring n-node Δ-regular graphs cannot be solved deterministically in ASYNC LOCAL.*

We complete the lower bound analysis with some additional results. The version of MIS considered in [16], which was proved impossible to solve, asks the nodes to output a set of vertices which forms an MIS in the graph induced by the *correct* nodes. Instead, we consider a weaker variant of MIS, asking the nodes to output a set of vertices which forms an MIS in the graph whenever all processes are correct, i.e., no crashes occurred. We show that even this weaker variant of MIS is impossible in ASYNC LOCAL.

▶ **Theorem 9.** *For every $n \geq 7$, no deterministic algorithms can solve weak MIS in the n-node cycle under ASYNC LOCAL.*

Finally, we show impossibility results for coloring general graphs.

▶ **Theorem 10.** *For every $\Delta \geq 2$, no deterministic algorithms can solve $(\Delta + 1)$-coloring in trees of maximum degree Δ under ASYNC LOCAL.*

We conclude, in Section 7, with some open questions.

4 Coloring General Graphs with $O(\Delta^2)$ Colors

In this section, we provide a simple algorithm for coloring a graph with $O(\Delta^2)$ colors. This algorithm is an adaptation of Linial's coloring algorithm [23] (which is designed to work in the LOCAL model) to the asynchronous setting. More in detail, we prove the following result.

▶ **Theorem 1.** *For every $\Delta \geq 2$, the round-complexity of $O(\Delta^2)$-coloring graphs of maximum degree Δ in the ASYNC LOCAL model is $O(\log^* n)$.*

In order to prove this result, we start by summarizing Linial's coloring algorithm, and then we show how to adapt it to the wait-free setting. We start by recalling the notion of set systems and of cover-free family of sets.

▶ **Definition 11.** *A* set system *is a pair $(X, \mathcal{F})$, where X is a set, and $\mathcal{F}$ is a collection of subsets of X. A set system $(X, \mathcal{F})$ is a* k-cover-free *family if, for every choice of $k+1$ distinct sets $S_0, S_1, \ldots, S_k$ in $\mathcal{F}$, the following holds: $S_0 \setminus \bigcup_{i=1}^{k} S_i \neq \varnothing$.*

To provide an intuition about how to use these two definitions, let us assume that the nodes of the input graph G are properly c-colored, and let us assume that there exists a Δ-cover-free family $(X, \mathcal{F})$ satisfying $c \leq |\mathcal{F}|$. It follows from these assumptions that there exists a one-to-one function f from the set of colors to $\mathcal{F}$. W.l.o.g., assume that X contains the numbers in $\{1, \ldots, |X|\}$. One step of Linial's algorithm is able to recolor the nodes with $c' = |X|$ colors, as follows.

1. Every node v communicates with its d neighbors to get their current colors $c_1, \ldots, c_d$, where $d \leq \Delta$ is the degree of v.
2. Every node v computes $X_v = f(c_v) \setminus \bigcup_{i=1}^{d} f(c_i)$, where c_v is the color of v, and then recolors itself with the minimum value in X_v.

Note that X_v is guaranteed to be non-empty by the fact that $(X, \mathcal{F})$ is a Δ-cover-free family, and that the obtained color c'_v satisfies $1 \leq c'_v \leq c'$. Linial's coloring algorithm repeats this process multiple times, each time using a different cover-free family. The runtime and the resulting number of colors depend on the choice of cover-free families. We summarize the cover-free families used by Linial's algorithm in the following two lemma.

▶ **Lemma 12** ([23])**.** **(a)** *For any $c > \Delta$, there exists a Δ-cover-free family $(X, \mathcal{F})$ with $c \leq |\mathcal{F}|$, and $|X| \leq 5\lceil \Delta^2 \log c \rceil$.* **(b)** *There exists a Δ-cover-free family $(X, \mathcal{F})$ with $10\Delta^3 \leq |\mathcal{F}|$, and $|X| \leq (4\Delta + 1)^2$.*

In [23], Lemma 12 has been proved in a non-constructive way. However, it is possible to obtain a similar statement by using polynomials over finite fields [14]. We will use the above lemma as a black-box. However, the correctness of our algorithm will be independent from which specific cover-free family construction is used.

We now discuss how these cover-free families are used. Linial's algorithm, in its standard formulation for LOCAL, requires the nodes to be aware of an upper bound N on the size of the identifier space. At the first round, nodes recolor themselves by using $5\lceil \Delta^2 \log N \rceil$ colors, thanks to a cover-free family from Lemma 12(a) with parameter $c = N$. We denote by f_1 the one-to-one function used by the nodes to map their color to the elements of the cover-free family. At the second round, nodes use the cover-free family from Lemma 12(a) with parameter $c = 5\lceil \Delta^2 \log N \rceil$, from which they obtain a coloring that uses $5\lceil \Delta^2 \log(5\lceil \Delta^2 \log N \rceil) \rceil$ colors. We denote by f_2 the one-to-one function used by the nodes to map their color to the elements of the cover-free family. The nodes repeat this process multiple times, each time using a cover-free family from Lemma 12(a) with parameter c equal to the amount of colors obtained

Algorithm 2 $O(\Delta^2)$-coloring arbitrary graph. Code of node v: $\mathrm{id}_v \in \{1,\ldots,N\}$; $\mathrm{input}_v = N$.

```
procedure WaitFreeLinial($\mathrm{id}_v$,$\mathrm{input}_v$)
    $S \leftarrow (\mathrm{id}_v, \perp, \ldots \perp)$; ▷ $S$ is an array of length $T+1 = O(\log^\star N)$, and is the state $s$ of $v$
    for $i = 1$ to $T$ do
        $(s_1,\ldots,s_d) \leftarrow \mathrm{WriteSnapshot}(s)$
        $A \leftarrow \{s_j.S[i] \mid (j \in \{1,\ldots,d\}) \wedge (s_j.S[i] \neq \perp)\}$    ▷ $i$th entry of each array $s_j.S$
        $S[i+1] \leftarrow \min f_i(S[i]) \smallsetminus \bigcup_{a\in A} f_i(a)$
    end for
    return $s[T+1]$
end procedure
```

in the previous rounds. Linial proved that it takes $O(\log^* N)$ rounds to reach a coloring that uses at most $10\Delta^3$ colors. Since it is typically assumed that $N = \mathrm{poly}(n)$, the runtime is $O(\log^* n)$. At this point, the cover-free family from Lemma 12(b) is used to get a coloring that uses $(4\Delta+1)^2 = O(\Delta^2)$ colors.

Let us denote by T the number of rounds performed in total, including the last round that uses the family from Lemma 12(b) for reducing the number of colors to at most $(4\Delta+1)^2$. For $1 \leq i \leq T$, let f_i be the one-to-one function used by the nodes to map their colors to the elements of the cover-free family while executing the ith round of Linial's algorithm.

The Algorithm. Let us show that the approach used in Linial's LOCAL algorithm can be adapted to work in ASYNC LOCAL as well. We assume that input_v contains the same upper bound N on the range of identifiers. So, in particular, every node v can compute T as a function of input_v. The adaptation of Linial's coloring algorithm to ASYNC LOCAL is displayed as Algorithm 2. The main challenge when running Linial's algorithm in the ASYNC LOCAL model comes from the fact that a vertex v may be in the ith iteration of Linial's algorithm, while a neighbor u of v may be in iteration $j \neq i$. Nevertheless, we will prove that our adaptation of Linial's algorithm correctly handles these cases. The runtime of Algorithm 2 is clearly $O(\log^* n)$. The proof that Algorithm 2 is correct can be found in the full version.

5 Reducing the Colors to $(\Delta+1)(\Delta+2)/2$

In this section, we show that, at the cost of increasing the running time by an additive factor depending on Δ only, we can decrease the amount of colors from $O(\Delta^2)$ to $\frac{1}{2}(\Delta+1)(\Delta+2)$.

▶ **Theorem 2.** *For every $\Delta \geq 2$, the round-complexity of $\frac{1}{2}(\Delta+1)(\Delta+2)$-coloring graphs of maximum degree Δ in the ASYNC LOCAL model is $O(\log^* n) + f(\Delta)$, where f is a function depending on Δ only.*

The algorithm that we provide is a generalization to general graphs of the 6-coloring algorithm for cycles presented in [16], and restated in Algorithm 1. On a high-level, the algorithms works as follows. First, we compute an initial $O(\Delta^2)$-coloring of the nodes. Then, the final color of each node is given by a pair (a,b). This pair is computed by repeatedly updating the values of a and b until the pair is different from the pairs of the neighbors. The value of a is updated as a function of the a-values of the neighbors with larger initial color, while the value of b is updated as a function of the b-values of the neighbors with smaller initial color.

Algorithm 3 Reducing the number of colors from $O(\Delta^2)$ to $(\Delta+1)(\Delta+2)/2$.

```
procedure SaveColors(id_v, input_v)
    $x \leftarrow \text{input}_v$; $(a,b) \leftarrow (0,0)$        ▷ $x \in [O(\Delta^2)]$ is the original color of $v$
    repeat forever                                                 ▷ $s = (x,a,b)$ is the state of $v$
        $(s_1,\dots,s_d) \leftarrow \text{WriteSnapshot}(s)$
        if $(a,b) \notin \{(s_i.a, s_i.b) \mid (i \in \{1,\dots,d_v\}) \wedge (s_i \neq \bot)\}$ then return $(a,b)$
        else
            $a \leftarrow \min \mathbb{N} \setminus \{s_i.a \mid (i \in \{1,\dots,d_v\}) \wedge (s_i \neq \bot) \wedge (x < s_i.x)\}$
            $b \leftarrow \min \mathbb{N} \setminus \{s_i.b \mid (i \in \{1,\dots,d_v\}) \wedge (s_i \neq \bot) \wedge (x > s_i.x)\}$
        end if
    end repeat
end procedure
```

The algorithm. In order to prove Theorem 2, we first analyze the algorithm SaveColors, displayed as Algorithm 3. Given an $O(\Delta^2)$-coloring as input, this procedure produces a $((\Delta+1)(\Delta+2)/2)$-coloring, in $f(\Delta)$ rounds for some function f. Theorem 2 follows by running Algorithm WaitFreeLinialReduced below, in which if a node v is running SaveColors while some neighbor u of v is still running WaitFreeLinial, then v treats the memory of u as $\bot$.

```
procedure WaitFreeLinialReduced(id_v, input_v)
    $c_v \leftarrow$ WaitFreeLinial(id_v, input_v)
    return SaveColors(id_v, $c_v$)
end procedure
```

The proofs of correctness and runtime of Algorithm 3 can be found in the full version of the paper.

6 Saving One More Color

We now modify Algorithm 3 in order to save one additional color. This new algorithm, shown in Algorithm 4, allows us to establish the following theorem.

▶ **Theorem 3.** *For every $\Delta \geq 2$, the round-complexity of $(\frac{1}{2}(\Delta+1)(\Delta+2)-1)$-coloring graphs of maximum degree Δ in the ASYNC LOCAL model is $O(\log^* n) + f(\Delta)$, for some function f that only depends on Δ.*

An important consequence of this result is Corollary 4, that is, the existence of a 5-coloring algorithm for the cycles in the ASYNC LOCAL model. This result is original because the 5-coloring algorithm proposed in [16] has a bug (cf. Appendix B where we exhibit an instance of 5-coloring C_4 for which the algorithm in [16] does not terminate).

6.1 Intuition of the algorithm

We start by providing the high level idea of the algorithm. The algorithm that we provide is similar to Algorithm 3, and it exploits some special properties of the pairs (a,b) that it produces. Specifically, we modify Algorithm 3 such that, if a node outputs the pair $(\Delta,0)$, then none of its neighbors output the pair $(0,\Delta)$. In this case, we can identify the pairs $(\Delta,0)$ and $(0,\Delta)$ as the same color, reducing the amount of colors in use by one.

Notice that a node that outputs the pair $(\Delta,0)$ is necessarily a local minimum with respect to the node identifiers, and similarly a node that outputs the pair $(0,\Delta)$ is necessarily a local maximum. The problematic case of neighbors outputting both pairs $(\Delta,0)$ and $(0,\Delta)$

can therefore only happen when the both neighbors are local extrema. However, for such neighboring nodes to reach a state where they would output problematic pairs, some specific conditions must hold which can be handled by the nodes as a specific case.

More in detail, in such a situation, we make nodes flip their relative ordering: if node u is a local minimum and node v is a local maximum, then u will treat v as smaller when comparing their x variables, and v will treat u as larger. By flipping relative ordering, we are forcing neighboring local extrema with pairs $(\Delta, 0)$ and $(0, \Delta)$ to stop being local extrema, leading them to change their output pairs. This modification will affect the termination time, and hence we also need to introduce new terminating conditions.

6.2 Formal Description

The algorithm is displayed in Algorithm 4, but some of its functions are presented below.

Treating special pairs as equal. The first modification applied to Algorithm 3 is the following. In line 5, instead of directly using the pairs (a, b) of the node, and the pairs of its neighbors, we first map them by using the function MAP shown below. Observe that MAP behaves as the identity function for all pairs different from $(\Delta, 0)$, and it maps $(\Delta, 0)$ to $(0, \Delta)$. In this way, the algorithm behaves similarly as the original one, except that it forbids neighboring nodes with pairs $(0, \Delta)$ and $(\Delta, 0)$ to terminate, since after applying MAP, they are both mapped to $(0, \Delta)$, and hence they are treated as having the same pair.

```
procedure MAP($a$,$b$)
    if $(a, b) = (\Delta, 0)$ then return $(0, \Delta)$
    else return $(a, b)$
    end if
end procedure
```

A new ordering relation. In Algorithm 3, nodes exploit their variables x (that is, the given coloring) to determine an ordering relation between them. In the new algorithm, each node keeps an additional variable f, which is a set of identifiers. The semantic is the following. For two nodes u and v, if $u \in v.f$ or $v \in u.f$, then the ordering w.r.t. their variables x is flipped. We call an edge $\{u, v\}$ flipped whenever $u \in v.f$ or $v \in u.f$.

Let us define two auxiliary Boolean functions that are used by a node v to determine whether the ordering relation with a neighbor u should be considered flipped or not. These functions take as input the state s_v and s_u of the two (neighboring) nodes. The variable z, as will be shown in the algorithm, stores the identifier of the node.

```
procedure ISNOTFLIPPED($s_v$, $s_u$)
    return $(s_v \neq \bot) \wedge (s_u \neq \bot) \wedge (s_u.z \notin s_v.f) \wedge (s_v.z \notin s_u.f)$
end procedure
procedure ISFLIPPED($s_v$, $s_u$)
    return $(s_v \neq \bot) \wedge (s_u \neq \bot) \wedge \big((s_u.z \in s_v.f) \vee (s_v.z \in s_u.f)\big)$
end procedure
```

We are now ready to define the new ordering relation. For this purpose, we define two functions that, given the state s of the node, and the state s_i of its ith neighbors, return the neighbors that are considered smaller, and the neighbors that are considered larger, respectively.

```
procedure $\textsc{Smaller}(s,(s_1,\ldots,s_k))$
    return $\{i \in \{1,\ldots,k\} \mid \big((\textsc{IsNotFlipped}(s,s_i) \wedge (s.x > s_i.x)\big)$
                                    $\vee\big(\textsc{IsFlipped}(s,s_i) \wedge (s.x < s_i.x)\big)\}$
end procedure
procedure $\textsc{Larger}(s,(s_1,\ldots,s_k))$
    return $\{i \in \{1,\ldots,k\} \mid \big(\textsc{IsNotFlipped}(s,s_i) \wedge (s.x < s_i.x)\big)$
                                    $\vee\big(\textsc{IsFlipped}(s,s_i) \wedge (s.x > s_i.x)\big)\}$
end procedure
```

Special termination. We also define a function that provides an extra termination condition. It relies on an additional function that detects a neighborhood with special properties. It uses some variables α and β that are both set to true if a node has at least one smaller neighbor (that is, it is not a local minima), and it has at least one larger neighbor (that is, it is not a local maxima). We assume that the maximum degree Δ is part of the input provided to the nodes.

```
procedure $\textsc{SpecialNeighborhood}(s,(s_1,\ldots,s_\Delta))$
    return $\Big(\big(\bigwedge_{i=1}^{\Delta}(s_i \neq \bot)\big) \wedge \big(\{s.a,s.b\} \cup (\cup_{i=1}^{\Delta}\{s_i.a,s_i.b\}) \subseteq \{0,\ldots,\Delta-1\}\big) \wedge s.\alpha \wedge s.\beta$
        $\wedge \Big(\bigwedge_{i=1}^{\Delta}\big((s_i.\alpha \vee |\textsc{Smaller}(s_i,[s])| = 1) \wedge (s_i.\beta \vee |\textsc{Larger}(s_i,[s])| = 1)\big)\Big)\Big)$
end procedure
```

That is, a neighborhood of a node v is *special* if (1) node v has seen all its neighbors, (2) they are precisely Δ, (3) the a and b variables of the node and of all its neighbors are in $\{0,\ldots,\Delta-1\}$, and (4) node v and all its neighbors have at least one smaller, and at least one larger neighbor. The reason why we use the condition $s_i.\alpha \vee |\textsc{Smaller}(s_i,[s])| = 1$ for checking whether a node has at least one smaller neighbor, instead of just using $s_i.\alpha$ is the following. Let u be the node with state s_i, and v be the node with state s. It could be the case that v is smaller than u, but u has been scheduled earlier than v. So it may be the case that u has never seen v. In this case, we could get that $u.\alpha$ is false, even though u has v as smaller neighbor. For this reason, node v computes whether $u.\alpha$ would become true if u were to be scheduled one additional round, by checking whether v is smaller than u using the condition $|\textsc{Smaller}(s_i,[s])| = 1$. A similar reasoning is applied for checking whether a node has at least one larger neighbor. Note that, in the algorithm, once a node sets α (resp. β) to true, that is when it realizes that it is not a local minima (resp., maxima), it will never change its value. The reason is that, as we will prove later, a node never becomes a local minima (resp., maxima) by flipping edges. We now introduce the special termination condition. According to this special condition, a node terminates if (1) its neighborhood is special, and (2) it is a local maxima according to the original ordering, that is, before flipping any edge.

```
procedure $\textsc{SpecialTermination}(s,(s_1,\ldots,s_\Delta))$
    return $\textsc{SpecialNeighborhood}(s,(s_1,\ldots,s_\Delta)) \wedge \big(\forall i \in \{1,\ldots,\Delta\}, s.x > s_i.x\big)$
end procedure
```

The new algorithm. The algorithm is displayed as Algorithm 4. Like in the case of Algorithm 3, we assume that input$_v$ is the result of running $\textsc{WaitFreeLinial}$. Observe that the algorithm is similar to Algorithm 3, with only three exceptions. First, it identifies $(0,\Delta)$ with $(\Delta,0)$ when checking for termination at line 6. Second, it uses the custom ordering relation induced by the functions $\textsc{Smaller}$ and $\textsc{Larger}$ at lines 11 and 12. Third, it has an additional termination condition at line 17. The proof that Algorithm 4 is correct, and the analysis of its runtime can be found in the full version.

Algorithm 4 Saving 1 color from palette $[\frac{1}{2}(\Delta+1)(\Delta+2)]$. Algorithm of node v with color input_v.

```
procedure SaveOneMoreColor($\text{id}_v$,$\text{input}_v$)
    $a \leftarrow 0$; $b \leftarrow 0$ ; $x \leftarrow \text{input}_v$ ; $z \leftarrow \text{id}_v$
    $f \leftarrow \{\}$; $\alpha \leftarrow$ false; $\beta \leftarrow$ false          ▷ $s = (a, b, x, f, \alpha, \beta, z)$ is the state of node $v$
    repeat forever
        $(s_1, \ldots, s_\Delta) \leftarrow \text{WriteSnapshot}(s)$          ▷ if $d_v < \Delta$, we assume $s_i = \bot$, $\forall i > d_v$
        if $\text{Map}(a, b) \notin \{\text{Map}(s_i.a, s_i.b) \mid i \in \{1, \ldots, \Delta\} \wedge s_i \neq \bot\}$ then return $\text{Map}(a, b)$
        else
            if $(a = \Delta) \vee (b = \Delta)$ then          ▷ We compute the flipped edges.
                $f \leftarrow f \cup \big\{s_i.z \mid (i \in \{1, \ldots, \Delta\})(s_i \neq \bot) \wedge \big((s_i.a = \Delta) \vee (s_i.b = \Delta)\big)\big\}$
            end if
            $a \leftarrow \mathbb{N} \setminus \{s_i.a \mid i \in \text{Larger}(s, (s_1, \ldots, s_\Delta))\}$
            $b \leftarrow \mathbb{N} \setminus \{s_i.b \mid i \in \text{Smaller}(s, (s_1, \ldots, s_\Delta))\}$
            if $|\text{Smaller}(s, (s_1, \ldots, s_\Delta))| \geq 1$ then $\alpha \leftarrow$ true
            end if
            if $|\text{Larger}(s, (s_1, \ldots, s_\Delta))| \geq 1$ then $\beta \leftarrow$ true
            end if
            if SpecialTermination$(s, (s_1, \ldots, s_\Delta))$ then return $(0, \Delta)$
            end if
        end if
    end repeat
end procedure
```

7 Open Questions

We have shown that every n-node graph of maximum degree Δ can be properly colored with $\frac{1}{2}(\Delta+1)(\Delta+2) - 1$ colors in ASYNC LOCAL, in $O(\log^\star n) + f(\Delta)$ rounds. The number of colors may seem large, but the ASYNC LOCAL model is considerably weaker than the (synchronous and failure-free) LOCAL model. In particular, it is known that even the clique with $n = \Delta + 1$ nodes cannot be colored with less than $2\Delta + 1$ colors in ASYNC LOCAL (whenever $\Delta + 1$ is power of a prime), and we have shown that there exists an infinite family of regular graphs with even degree Δ that cannot be colored with less than $\Delta + 3$ colors in ASYNC LOCAL. One major question as far as solving graph problems in asynchronous crash-prone networks is thus the following.

Open Problem: Is there a $(2\Delta + 1)$-coloring algorithm for graphs with maximum degree Δ in the ASYNC LOCAL model, for every $\Delta \geq 2$?

Of course, if one puts aside cliques, there might be a coloring algorithm for ASYNC LOCAL using a palette of less than $2\Delta + 1$ colors. However, we have shown that, for $\Delta = 2$, the bound $2\Delta + 1 = 5$ is tight for infinitely many cycles. The only generic bound applying to infinitely many graphs of maximum degree Δ is however only $\Delta + 3$, so there might be room for improvement. Yet, saving even just a single color in a palette of $\frac{1}{2}(\Delta+1)(\Delta+2)$ colors was very delicate and difficult. So, progressing from a quadratic number of colors to a linear number of colors appears to be a challenge in ASYNC LOCAL.

Finally, we question the efficiency of randomized algorithms in the ASYNC LOCAL model.

Open Problem: To which extent randomized algorithms help in the ASYNC LOCAL model, in term of both complexity and computability?

References

1 Hagit Attiya and Ami Paz. Counting-based impossibility proofs for set agreement and renaming. *J. Parallel Distributed Comput.*, 87:1–12, 2016. doi:10.1016/J.JPDC.2015.09.002.

2 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics.* Wiley, 2004.

3 Baruch Awerbuch, Boaz Patt-Shamir, David Peleg, and Michael E. Saks. Adapting to asynchronous dynamic networks. In *24th ACM Symposium on Theory of Computing (STOC)*, pages 557–570, 1992.

4 Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *31st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 514–522, 1990. doi:10.1109/FSCS.1990.89572.

5 Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed $(\delta + 1)$-coloring below szegedy-vishwanathan barrier, and applications to self-stabilization and to restricted-bandwidth models. In *37^{th} ACM Symposium on Principles of Distributed Computing (PODC)*, pages 437–446, 2018. URL: https://dl.acm.org/citation.cfm?id=3212769.

6 Samuel Bernard, Stéphane Devismes, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Optimal deterministic self-stabilizing vertex coloring in unidirectional anonymous networks. In *23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–8, 2009. doi:10.1109/IPDPS.2009.5161053.

7 Jean R. S. Blair and Fredrik Manne. An efficient self-stabilizing distance-2 coloring algorithm. *Theoretical Computer Science*, 444:28–39, 2012. doi:10.1016/J.TCS.2012.01.034.

8 Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder. Brief announcement: Memory lower bounds for self-stabilization. In *33rd International Symposium on Distributed Computing (DISC)*, volume 146 of *LIPIcs*, pages 37:1–37:3. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.DISC.2019.37.

9 Lélia Blin and Sébastien Tixeuil. Compact deterministic self-stabilizing leader election - the exponential advantage of being talkative. In *27th Int. Symp. on Distributed Computing (DISC)*, volume 8205 of *LNCS*, pages 76–90. Springer, 2013. doi:10.1007/978-3-642-41527-2_6.

10 Armando Castañeda, Pierre Fraigniaud, Ami Paz, Sergio Rajsbaum, Matthieu Roy, and Corentin Travers. Synchronous t-resilient consensus in arbitrary graphs. *Inf. Comput.*, 292:105035, 2023. doi:10.1016/J.IC.2023.105035.

11 Armando Castañeda, Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Michel Raynal. Making local algorithms wait-free: the case of ring coloring. *Theory of Computing Systems*, 63(2):344–365, 2019. doi:10.1007/S00224-017-9772-Y.

12 Richard Cole and Uzi Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *18th ACM Symposium on Theory of Computing (STOC)*, pages 206–219, 1986. doi:10.1145/12130.12151.

13 Carole Delporte-Gallet, Hugues Fauconnier, Pierre Fraigniaud, and Mikaël Rabie. Distributed computing in the asynchronous LOCAL model. In *21st International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS 11914, pages 105–110. Springer, 2019.

14 P. Erdös, P. Frankl, and Z. Füredi. Families of finite sets in which no set is covered by the union ofr others. *Israel Journal of Mathematics*, 51(1):79–89, 1985.

15 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

16 Pierre Fraigniaud, Patrick Lambein-Monette, and Mikaël Rabie. Fault tolerant coloring of the asynchronous cycle. In *36th Int. Symp. on Distributed Computing (DISC)*, volume 246 of *LIPIcs*, pages 23:1–23:22. Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.DISC.2022.23.

17 Marc Fuchs and Fabian Kuhn. List defective colorings: Distributed algorithms and applications. In *37th Int. Symp. on Distributed Computing (DISC)*, volume 281 of *LIPIcs*, pages 22:1–22:23. Schloss Dagstuhl - Leibniz-Zentrum für Inf., 2023. doi:10.4230/LIPICS.DISC.2023.22.

18 Mohsen Ghaffari and Fabian Kuhn. Deterministic distributed vertex coloring: Simpler, faster, and without network decomposition. In *62nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1009–1020, 2021. doi:10.1109/FOCS52979.2021.00101.

19 Mohsen Ghaffari and Anton Trygub. A near-optimal deterministic distributed synchronizer. In *42th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 180–189, 2023. doi:10.1145/3583668.3594598.

20 Magnús M. Halldórsson, Fabian Kuhn, Alexandre Nolin, and Tigran Tonoyan. Near-optimal distributed degree+1 coloring. In *54th ACM Symposium on Theory of Computing (STOC)*, pages 450–463, 2022. doi:10.1145/3519935.3520023.

21 Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology.* Morgan Kaufmann, 2013.

22 Juho Hirvonen and Jukka Suomela. *Distributed Algorithms.* Creative Commons, 2020.

23 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

24 Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

25 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

26 Thomas Nowak, Ulrich Schmid, and Kyrill Winkler. Topological characterization of consensus under general message adversaries. In *38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 218–227, 2019. doi:10.1145/3293611.3331624.

27 David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, 2000.

28 Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach.* Springer, 2018. doi:10.1007/978-3-319-94141-7.

29 Nicola Santoro and Peter Widmayer. Time is not a healer. In *6th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 349 of *LNCS*, pages 304–313. Springer, 1989. doi:10.1007/BFB0028994.

30 Kyrill Winkler, Ami Paz, Hugo Rincon Galeana, Stefan Schmid, and Ulrich Schmid. The time complexity of consensus under oblivious message adversaries. In *14th Innovations in Theoretical Computer Science Conference (ITCS)*, volume 251 of *LIPIcs*, pages 100:1–100:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ITCS.2023.100.

A Example of an execution of an algorithm for 6-coloring cycles

An example of an execution of Algorithm 1 is provided in Table 1, in which the old and new states of each node after each step is displayed.

Table 1 An example of execution of Algorithm 1, for the cycle C_5 with consecutive node identifiers $(3, 5, 4, 1, 6)$. The example corresponds to the scheduling $\mathcal{S} = \{1, 3, 5\}, \{4, 5\}, \{3, 4\}, \{6\}, \{6\}, \dots,$ and T stands for Terminated. At each step, the states that are updated are highlighted in bold.

	3		5		4		1		6	
	Old	New	Old	New	Old	New	Old	New	Old	New
Initialization	$\perp$	$(3,0,0)$	$\perp$	$(5,0,0)$	$\perp$	$(4,0,0)$	$\perp$	$(1,0,0)$	$\perp$	$(6,0,0)$
$\{1,3,5\}$										
after write	$\mathbf{(3,0,0)}$	$(3,0,0)$	$\mathbf{(5,0,0)}$	$(5,0,0)$	$\perp$	$(4,0,0)$	$\mathbf{(1,0,0)}$	$(1,0,0)$	$\perp$	$(6,0,0)$
update	$(3,0,0)$	$\mathbf{(3,1,0)}$	$(5,0,0)$	$\mathbf{(5,0,1)}$	$\perp$	$(4,0,0)$	$(1,0,0)$	$\boldsymbol{T(0,0)}$	$\perp$	$(6,0,0)$
$\{4,5\}$										
after write	$(3,0,0)$	$(3,1,0)$	$\mathbf{(5,0,1)}$	$(5,0,1)$	$\mathbf{(4,0,0)}$	$(4,0,0)$	$(1,0,0)$	$T(0,0)$	$\perp$	$(6,0,0)$
update	$(3,0,0)$	$(3,1,0)$	$(5,0,1)$	$\boldsymbol{T(0,1)}$	$(4,0,0)$	$\mathbf{(4,1,1)}$	$(1,0,0)$	$T(0,0)$	$\perp$	$(6,0,0)$
$\{3,4\}$										
after write	$\mathbf{(3,1,0)}$	$(3,1,0)$	$(5,0,1)$	$T(0,1)$	$\mathbf{(4,1,1)}$	$(4,1,1)$	$(1,0,0)$	$T(0,0)$	$\perp$	$(6,0,0)$
update	$(3,1,0)$	$\boldsymbol{T(1,0)}$	$(5,0,1)$	$T(0,1)$	$(4,1,1)$	$\boldsymbol{T(1,1)}$	$(1,0,0)$	$T(0,0)$	$\perp$	$(6,0,0)$
$\{6\}$										
after write	$(3,1,0)$	$T(1,0)$	$(5,0,1)$	$T(0,1)$	$(4,1,1)$	$T(1,1)$	$(1,0,0)$	$T(0,0)$	$\mathbf{(6,0,0)}$	$(6,0,0)$
update	$(3,1,0)$	$T(1,0)$	$(5,0,1)$	$T(0,1)$	$(4,1,1)$	$T(1,1)$	$(1,0,0)$	$T(0,0)$	$(6,0,0)$	$\mathbf{(6,0,1)}$
$\{6\}$										
after write	$(3,1,0)$	$T(1,0)$	$(5,0,1)$	$T(0,1)$	$(4,1,1)$	$T(1,1)$	$(1,0,0)$	$T(0,0)$	$\mathbf{(6,0,1)}$	$(6,0,1)$
update	$(3,1,0)$	$T(1,0)$	$(5,0,1)$	$T(0,1)$	$(4,1,1)$	$T(1,1)$	$(1,0,0)$	$T(0,0)$	$(6,0,1)$	$\boldsymbol{T(0,1)}$

B A Counterexample for an Existing Algorithm for 5-Coloring Cycles

We merely exhibit an instance of 5-coloring C_4 for which the algorithm in [16] does not terminate[4]. The algorithm presented in [16] is shown in Algorithm 5. In Table 2, we provide an example of execution where the algorithm loops forever.

[4] For the interested reader, we found this counterexample by implementing a simulator for the ASYNC LOCAL model. This simulator tests a given algorithm with random schedulings.

Algorithm 5 The (erroneous) 5-coloring algorithm of [16].

```
procedure FiveColoring(id_v,input_v)
    $x \leftarrow \mathrm{id}_v$;   $a \leftarrow 0$;   $b \leftarrow 0$                 ▷ $s = (x, a, b)$ is the state of node $v$
    repeat forever
        $(s_1, s_2) \leftarrow \mathrm{WriteSnapshot}(s)$
        $P^+ \leftarrow \{i \in \{1,2\} \mid s_i \neq \bot \wedge s_i.x > x\}$          ▷ neighbors with larger id
        $C^+ \leftarrow \{x_i.a \mid i \in P^+\} \cup \{x_i.b \mid i \in P^+\}$     ▷ $a$ and $b$ of neighbors with larger id
        $C \leftarrow \{x_i.a \mid i \in \{1,2\} \wedge s_i \neq \bot\} \cup \{x_i.b \mid i \in \{1,2\} \wedge s_i \neq \bot\}$   ▷ $a$ and $b$ of all
neighbors
        if $a \notin C$ then return $a$
        else
            if $b \notin C$ then return $b$
            else
                $a \leftarrow \min \mathbb{N} \setminus C^+$
                $b \leftarrow \min \mathbb{N} \setminus C$
            end if
        end if
    end repeat
end procedure
```

Table 2 An example of execution where Algorithm 5 loops, for a 4-cycle with nodes' identifiers $(3,4,2,1)$ in consecutive order. The example is for the scheduling $\{2,3,4\}, \{1,3,4\}, \{3,4\}, \{3,4\}, \ldots$. Observe that the state obtained after scheduling $\{1,3,4\}$ is the same state as the one obtained after the fourth step (when $\{3,4\}$ is scheduled for the second time). Therefore, there exists a scheduling that makes the algorithm looping forever.

	3		4		2		1	
	Old	New	Old	New	Old	New	Old	New
Initialization	$\bot$	$(3,0,0)$	$\bot$	$(4,0,0)$	$\bot$	$(2,0,0)$	$\bot$	$(1,0,0)$
$\{2,3,4\}$								
after write	$\mathbf{(3,0,0)}$	$(3,0,0)$	$\mathbf{(4,0,0)}$	$(4,0,0)$	$\mathbf{(2,0,0)}$	$(2,0,0)$	$\bot$	$(1,0,0)$
update	$(3,0,0)$	$\mathbf{(3,1,1)}$	$(4,0,0)$	$\mathbf{(4,0,1)}$	$(2,0,0)$	$\mathbf{(2,1,1)}$	$\bot$	$(1,0,0)$
$\{1,3,4\}$								
after write	$\mathbf{(3,1,1)}$	$(3,1,1)$	$\mathbf{(4,0,1)}$	$(4,0,1)$	$(2,0,0)$	$(2,1,1)$	$\mathbf{(1,0,0)}$	$(1,0,0)$
update	$(3,1,1)$	$\mathbf{(3,2,2)}$	$(4,0,1)$	$\mathbf{(4,0,2)}$	$(2,0,0)$	$(2,1,1)$	$(1,0,0)$	$\mathbf{(1,2,2)}$
$\{3,4\}$								
after write	$\mathbf{(3,2,2)}$	$(3,2,2)$	$\mathbf{(4,0,2)}$	$(4,0,2)$	$(2,0,0)$	$(2,1,1)$	$(1,0,0)$	$(1,2,2)$
update	$(3,2,2)$	$\mathbf{(3,1,1)}$	$(4,0,2)$	$\mathbf{(4,0,1)}$	$(2,0,0)$	$(2,1,1)$	$(1,0,0)$	$(1,2,2)$
$\{3,4\}$								
after write	$\mathbf{(3,1,1)}$	$(3,1,1)$	$\mathbf{(4,0,1)}$	$(4,0,1)$	$(2,0,0)$	$(2,1,1)$	$(1,0,0)$	$(1,2,2)$
update	$(3,1,1)$	$\mathbf{(3,2,2)}$	$(4,0,1)$	$\mathbf{(4,0,2)}$	$(2,0,0)$	$(2,1,1)$	$(1,0,0)$	$(1,2,2)$

Speedup of Distributed Algorithms for Power Graphs in the CONGEST Model

Leonid Barenboim
Open University of Israel, Ra'anana, Israel

Uri Goldenberg
Ben-Gurion University of the Negev, Beersheba, Israel

Abstract

We obtain improved distributed algorithms in the CONGEST message-passing setting for problems on power graphs of an input graph G. This includes Coloring, Maximal Independent Set, and related problems. For $R = f(\Delta^k, n)$, we develop a general deterministic technique that transforms R-round LOCAL model algorithms for G^k with certain properties into $O(R \cdot \Delta^{k/2-1})$-round CONGEST algorithms for G^k. This improves the previously-known running time for such transformation, which was $O(R \cdot \Delta^{k-1})$. Consequently, for problems that can be solved by algorithms with the required properties and within polylogarithmic number of rounds, we obtain *quadratic* improvement for G^k and *exponential* improvement for G^2. We also obtain significant improvements for problems with larger number of rounds in G. Notable implications of our technique are the following deterministic distributed algorithms:

- We devise a distributed algorithm for $O(\Delta^4)$-coloring of G^2 whose number of rounds is $O(\log \Delta + \log^* n)$. This improves exponentially (in terms of Δ) the best previously-known deterministic result of Halldorsson, Kuhn and Maus.[25] that required $O(\Delta + \log^* n)$ rounds, and the standard simulation of Linial [30] algorithm in G^k that required $O(\Delta \cdot \log^* n)$ rounds.
- We devise an algorithm for $O(\Delta^2)$-coloring of G^2 with $O(\Delta \cdot \log \Delta + \log^* n)$ rounds, and $(\Delta^2 + 1)$-coloring with $O(\Delta^{1.5} \cdot \log \Delta + \log^* n)$ rounds. This improves quadratically, and by a power of $4/3$, respectively, the best previously-known results of Halldorsson, Khun and Maus. [25].
- For $k > 2$, our running time for $O(\Delta^{2k})$-coloring of G^k is $O(k \cdot \Delta^{k/2-1} \cdot \log \Delta \cdot \log^* n)$.
  Our running time for $O(\Delta^k)$-coloring of G^k is $\tilde{O}(k \cdot \Delta^{k-1} \cdot \log^* n)$.
  This improves best previously-known results quadratically, and by a power of $3/2$, respectively.
- For constant $k > 2$, our upper bound for $O(\Delta^{2k})$-coloring of G^k nearly matches the lower bound of Fraigniaud, Halldorsson and Nolin. [16] for *checking* the correctness of a coloring in G^k.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Mathematics of computing → Graph coloring

Keywords and phrases Distributed Algorithms, Graph Coloring, Power Graph, CONGEST

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.6

Related Version *Full Version*: https://arxiv.org/abs/2305.04358 [10]

Funding This research was supported by The Open University of Israel's Research Fund.

1 Introduction

1.1 Model and Results

In the distributed message-passing model a communication network is represented by an unweighted n-vertex graph $G = (V, E)$ with maximum degree Δ. Each vertex has a unique ID, represented by $O(\log n)$ bits. Computations proceed in synchronous discrete rounds, each of which consists of message exchange between neighbours, and local computations of vertices. The input for an algorithm is the network graph G, where initially each vertex

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 6; pp. 6:1–6:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

knows only its own ID and the IDs of its neighbours[1], and the values n and Δ. During an execution, within $k > 0$ rounds, vertices may obtain information about other vertices in their k-hop-neighborhood. The output of an algorithm consists of the final answers returned by all vertices. The running time of an algorithm is the number of rounds until the algorithm terminates in all vertices. In the current paper we focus on the *CONGEST* model. In this model the number of bits that can be sent over each edge in each round is bounded by $O(\log n)$. Consequently, within k rounds each vertex can learn only a small portion of the information that resides in its k-hop-neighborhood. Such a neighborhood may contain up to $\Delta^k + 1$ vertices, whose information is much larger than what can be received in k rounds, namely up to $O(k \cdot \Delta \cdot \log n)$ bits. This is in contrast to the distributed *LOCAL* model, where message size is unbounded, and each vertex can learn the entire k-hop-neighborhood within k rounds.

Among the most studied problems in this setting are coloring and maximal independent set (henceforth, MIS). These problems are very well motivated by real-life network tasks, such as resource allocation, scheduling, channel assignment, etc. Often, in order to perform such a task, a coloring or MIS has to be computed on a *power graph* of the network, rather than the original graph that represents the network. For example, in job scheduling, where each vertex can perform one job at a time, and can send one neighbor a job to perform, a coloring is used for job scheduling. In this scheduling all vertices of a certain color are executed at the same time. Then these vertices and neighbours selected by them execute jobs. However, an ordinary proper coloring will not suffice, since a vertex may have several neighbours with the same color, who send it jobs to perform. Since it can handle only one job, the other jobs that are sent simultaneously are lost. To prevent this, 2-distance coloring is used, where each pair of vertices at distance at most 2 one from another in G obtain distinct colors. (This is equivalent to an ordinary proper coloring of the power graph G^2.) Now each vertex has at most one neighbor with a certain color, and no more than one job arrives to the vertex at a time.

Because of their importance, problems on power graphs for computing coloring, MIS, and related tasks have been very intensively studied in the distributed setting. A plethora of significant results have been obtained in recent years in the CONGEST setting [4, 13, 16, 25, 26, 33]. In particular, Bar-Yehuda, Censor-Hillel, Maus, Pai and Pemmaraju [4] devised approximate Minimum Dominating Set and Minimum Vertex cover (MVC) algorithms for G^2. Halldorsson, Khun, Maus and Nolin. [26] devised a logarithmic-time randomized algorithm for distance-2 coloring using $\Delta^2 + 1$ colors. (For a positive integer q, distance-2 coloring with q colors is equivalent to ordinary proper coloring of G^2 with q colors.) Halldorsson, Khun and Maus [25] devised deterministic algorithms for distance-2 coloring using $O(\Delta^2)$ colors with time polylog(n), and using $(\Delta^2 + 1)$ colors with time $O(\Delta^2 + \log^* n)$. Fraigniaud, Halldorsson and Nolin [16] showed that for $k > 2$, testing whether a given proper coloring is correct requires $\Omega(\Delta^{\lfloor (k-1)/2 \rfloor})$ rounds. Also, a general well-known scheme for simulating LOCAL R-round algorithms for G^k in the CONGEST setting provides algorithms with time $O(R \cdot \Delta^{k-1})$ in the CONGEST setting.

In the last decades, much attention of researchers was devoted to understanding the complexity of these problems on G and G^k as a function of Δ, modulo the unavoidable factor of $\log^* n$. A major question that remained open for many years is whether $o(\Delta) + \log^* n$ solutions are possible for $O(\Delta)$-coloring, MIS and related problems on G. In recent breakthroughs it was shown that while MIS and Maximal Matching require $\Omega(\Delta)$ time [3], the problem

[1] A node can learn the ids of its neighbours within a single round in the CONGEST model

of $O(\Delta)$-coloring can be solved in $\tilde{O}(\sqrt{\Delta} + \log^* n)$ time [5]. However, this question for coloring power graphs in sublinear-in-Δ^k time still remained open, since the best results for $O(\Delta^k)$-coloring of G^k were at least $O(\Delta^k)$. Specifically, for $k = 2$ it is $O(\Delta^2 + \log^* n)$ [25], and for $k > 2$ it is $\tilde{O}(\Delta^{k+k/2-1} + \Delta^{k-1} \log^* n)$ (by applying a standard simulation for G^k to [9]).

In this paper we answer this question in the affirmative, by providing deterministic CONGEST algorithms for $O(\Delta^2)$-coloring of G^2 with $\tilde{O}(\Delta + \log^* n)$ rounds, and $O(\Delta^k)$-coloring of G^k with $\tilde{O}(\Delta^{k-1} + \log^* n)$ rounds[2]. More generally, we provide a speedup technique for various problems, including coloring and MIS, that improves quadratically each phase of an algorithm for G^k that adheres to certain requirements. In particular, such algorithms that perform the standard simulation are improved from running time $O(R \cdot \Delta^{k-1})$ to $O(R \cdot \Delta^{k/2-1})$. In the case of G^2 our speedup of a phase is *exponential* (in terms of Δ), resulting in running time $O(R \cdot \text{polylog}(\Delta))$. For example, we compute $O(\Delta^4)$-coloring of G^2 within $O(\log \Delta + \log^* n)$ time, improving the best previously-known result of $O(\Delta + \log^* n)$.

Our results also give rise to a quadratic improvement in the *memory complexity* per vertex. Specifically, when using *aggregation functions*, the size of the result computed by w for each vertex in its $k/2$-hop-neighborhood is reduced from $\Delta^{k/2}$ to a much smaller value, ideally, $O(1)$. Using such technique the required memory per vertex is also reduced from $\tilde{O}(\Delta^k)$ to $\tilde{O}(\Delta^{k/2})$.

An interesting implication of our results is that $O(\Delta^{2k})$-coloring of G^k can be computed in the CONGEST model in $O(\Delta^{k/2-1} \log \Delta \log^* n)$ rounds. This nearly matches the lower bound of Fraigniaud, Halldorsson and Nolin [16] for testing a proper coloring that requires $\Omega(\Delta^{\lfloor (k-1)/2 \rfloor})$ rounds.

1.2 Our Techniques

The previously-known technique for simulating CONGEST algorithms from G on G^k proceeds as follows. For each round of the simulated algorithm, each vertex $v \in V$ has to obtain its k-hop-neighborhood information. Since the number of vertices in the k-hop-neighborhood is bounded by $O(\Delta^k)$ and the number of edges is bounded by $O(\Delta^k \cdot \Delta) = O(\Delta^{k+1})$, this neighborhood structure (consisting of vertices and edges between them) can be delivered to v within $O(\Delta^k)$ rounds. This information is sufficient for v to simulate its local computation in that round for the algorithm in G^k. Then, v broadcasts a message to all vertices in its k-hop-neighborhood. Such a broadcast is performed by all vertices of G in parallel, and requires $O(\Delta^{k-1})$ rounds. We note that often it is sufficient to employ $O(\Delta^{k-1})$ rounds also in the stage of obtaining the k-hop-neighborhood information. This is the case when only information about vertices is needed, rather than how they are connected. Consequently, various CONGEST algorithms that require $f(\Delta, n)$ rounds in G can be transformed into $O(f(\Delta^k, n) \cdot \Delta^{k-1})$-round algorithms for G^k.

Our new method improves this idea, by performing stages of information collection and information broadcast only half the way, to distance $k/2$ rather than k. Indeed, for each pair of vertices $u, v \in V$ at distance k one from another, there is a vertex w in the middle of a path between them that can obtain their information by collecting its $k/2$-hop-neighborhood. Then w can perform a computation on v and w (as well as all other vertices in its $k/2$-hop-neighborhood) and return the results to them. However, if this is done in a trivial way, then w

[2] For simplicity of presentation, we assume that k is even. Our results extend directly to any positive integer $k \geq 2$, by replacing $k/2$ with $\lceil k/2 \rceil$.

should return information of size $\Delta^{k/2}$ to each vertex in its $k/2$-hop-neighborhood. In general, this is done not only by w, but by all vertices in the graph in parallel. Consequently, in order to collect $\Delta^{k/2}$ pieces of information from each of $\Delta^{k/2}$ vertices in a $k/2$-hop-neighborhood, again $O(\Delta^k)$ rounds are needed. Thus, a straightforward approach for going half the way does not provide an improvement. But we obtain an improvement using the following more sophisticated method.

Various algorithms can be decomposed into basic steps that perform such operations as checking whether a variable of a vertex appears in its k-hop-neighborhood, summing the number of certain value, computing maximum or minimum, etc. In such cases, rather than collecting the entire k-hop-neighborhood information and then computing a function locally, aggregation functions can be applied iteratively and distributively. For example, consider a function for computing the maximum of variables in the k-hop-neighborhood. In this case, w who is in the middle of a k-length path between u and v applies the function iteratively and locally on all vertices in the $k/2$-hop-neighborhood of w. The vertex w sends the result to all its immediate neighbours, including the one that is at distance $k/2 - 1$ from v. Then this vertex computes the maximum of the maxima it received from its neighbours, and sends it to its own neighbours, one of which is at distance $k/2 - 2$ from v. After $k/2$ such rounds, v receive the maximum value in its k-hop-neighborhood.

The example of the maximum function is a simple case, when the maximum in the $k/2$-hop-neighborhood of w is the same for all its vertices. However, in general, different answers may be needed for different vertices. Another example is a function that checks whether variables of vertices are equal. It may be the case that u has a vertex in the $k/2$-hop-neighborhood of w with a variable that equals to that of u, but v does not have such a variable. Then, w must store for each vertex in its $k/2$-hop-neighborhood its own answer. This is done by w, by applying aggregation functions locally and iteratively, for each such vertex. The result of size $\Delta^{k/2}$ is sent to all neighbours of w in parallel. Then, each neighbor applies the aggregation functions for all vertices in its $(k/2 - 1)$-hop-neighborhood. It does so starting from the results it received from w. This way, the outcome now is regarding distance $k/2 + 1$. This continues for $k/2$ phases, where in each phase less data has to be sent (it is reduced by a factor of Δ in each phase), but the radius of the computation grows by 1. After $k/2$ such phases, each vertex holds the result of function applications in its k-hop-neighborhood.

In order for this technique to work, it employs functions that are (1) *commutative* and (2) *idempotent*. That is, (1) the order of function application must not affect the result, and (2) applying the same function several times must not affect the result, no matter how many times it is applied. For example, this is the case in the function $max_t(x) = max(x, t)$. A series of applications $max_{t_1}(max_{t_2}(...(max_{t_q}(x)...))$ can be applied in any order, and each max_{t_i} can appear any positive number of times, without changing the outcome. This is important to the success of our method, since pairs of vertices u, v may belong to many $k/2$-hop-neighborhoods that perform computations for them, and the order of computations is not predefined. As mentioned above, we show that various algorithms for complicated tasks, including coloring and MIS, can be constructed from steps consisting of such operations.

We extend our technique also to functions that are not idempotent, in order to generalize it further. In particular, the counting operation is a very useful building block in various algorithms. For example, it makes it possible to compute how many times a certain value appears in the variables of a k-hop-neighborhood of a vertex. (The function increases the result by one each time it is applied on a variable with that value. But when the function is applied on a variable with a different value, the result does not change.) However, this operation is not idempotent, since the outcome depends on the number of times the function is applied. Consequently, if we apply our technique as described above, the result may be larger than the actual number of variables with that value. This is because a certain function invocation may be repeated several times by different vertices in the k-hop-neighborhood.

We propose two solutions for this challenge. In the first solution we analyze how much the result over-counts the correct answer. In certain cases this can be bounded, so an algorithm still works, even with over-counting. In more complicated cases, when precise computation is required, we use BFS trees of radius $k/2$ that are constructed from all vertices in parallel. These trees are broadcasted to distance $k/2$. Then unique paths can be produced between pairs of vertices at distance k one from another. These paths are used to make sure that each function application is executed exactly once, for an (ordered) pair of vertices in a k-hop-neighborhood. This incurs an additional running time of $O(\Delta^{k-1})$, but only once during execution. Consequently, the running time of a transformation of an $f(\Delta, n)$-round algorithm for G becomes $O(\Delta^{k-1} + f(\Delta^k, n) \cdot \Delta^{k/2-1})$ in G^k. This is again a significant improvement over the previously known time of $O(f(\Delta^k, n) \cdot \Delta^{k-1})$.

Another tool we introduce for shrinking message size, which may be of independent interest, is *binary search in neighborhoods.* A common building block of distributed algorithms is performing computations on lists that vertices hold, as follows. A vertex has to compute a certain function on its own list and the lists of its neighbours. In some functions the outcome can be determined by a single element of a list. (For example, finding an element that does not appear in the neighbours lists.) A naive computation for lists of size t requires t rounds, since the lists have to be delivered to neighbours that apply the function on them. To speedup this process we perform a binary search with assistance of neighbours, so that lists shrink by a certain factor in each round, but each of them still contains an element from which the function outcome can be deduced. Finally, each list contains just a single element, who provides the desired result. This tool is a main ingredient in our algorithm for $O(\Delta^4)$-coloring of G^2 in $O(\log \Delta + \log^* n)$ time, and $O(\Delta^2)$-coloring of G^2 in $O(\Delta \cdot \log \Delta + \log^* n)$ time.

1.3 Related Work

Among the first works on deterministic distributed symmetry breaking are algorithms for (1-hop) coloring and MIS of paths and trees. An $O(\log^* n)$-round algorithm for 3-coloring paths was devised by Cole and Vishkin in 1986 [12]. This was extended to oriented trees by Goldberg, Plotkin, and Shannon in 1987 [24]. A (1-hop) coloring deterministic algorithm for general graphs that employs $O(\Delta^2)$-colors and has running time $O(\log^* n)$ was obtained by Linial in 1987 [30]. This algorithm gives rise to $(\Delta + 1)$-coloring in $O(\Delta^2 + \log^* n)$ time. The running time for $(\Delta + 1)$-coloring was improved to $O(\Delta \cdot \log \Delta + \log^* n)$ by Szegedy and Vishwanathan in 1993 [36], and by Kuhn and Wattenhofer in 2006 [29], by a more explicit construction. This was further improved to $O(\Delta + \log^*)$ by Barenboim, Elkin and Kuhn in 2009 [7]. The latter result also implies an algorithm for MIS in $O(\Delta + \log^* n)$ rounds. As proven in [3], this result for MIS is tight. On the other hand, sublinear-in-Δ algorithms for $(\Delta + 1)$-coloring are possible, as shown by Barenboim [5], who devised such an algorithm with running time $\tilde{O}(\Delta^{3/4} + \log^* n)$. This was further improved in several works [9, 17, 18, 32], to the current state of the art, which is $\tilde{O}(\sqrt{\Delta} + \log^* n)$.

In addition to the thread of research on algorithms with dependency on Δ in the running time, there has been also progress with deterministic algorithms that depend on n (in a stronger way than just $\log^* n$). Several results were obtained using network decompositions [2, 34, 21], and the recent breakthrough of Rozhon and Ghaffari [21] makes it possible to compute coloring and MIS deterministically within $\text{poly}(\log n)$ time. In addition, it is even possible to obtain $(\Delta + 1)$-coloring and MIS without network decompositions in $O(\log n \log^2 \Delta)$ time [23, 14] and $(\log^2 n)$ for MIS [20]. Both threads of research are very important and attracted much attention of researchers. In the case where the dependency on n is larger than $\log^* n$, a main goal is improving this dependency, as well as the dependency on Δ. In the case that the dependency on n is $O(\log^* n)$, which is unavoidable, the main goal is improving the dependency on Δ.

Randomized symmetry breaking algorithms have been very extensively studied as well. The first algorithms for $(\Delta + 1)$-coloring and MIS, due to Luby [31] and Alon, Babai and Itai [1] required $O(\log n)$-time. This was improved by Kothapalli, Scheideler, Onus and Schindelhauer in 2006 [28] who obtained $O(\Delta)$-coloring in $\tilde{O}(\sqrt{\log n})$ rounds. In several major advances, this was improved even further to $O(\log \Delta + \text{poly}(\log \log n))$ for MIS [19, 35], $O(\sqrt{\log \Delta} + \text{poly}(\log \log n))$ for coloring [27], and $O(\text{poly}(\log \log n))$ for coloring [11].

Since currently-known randomized algorithms have better dependency on Δ, while deterministic algorithms have better dependency on n, the improvement of either randomized or deterministic solutions is valuable. In particular, improving the dependency on Δ in deterministic algorithms is very important, since the gap between the current deterministic and randomized solutions is quite large.

Recently, distance-k problems and computations on power graphs attracted much attention in the research of distributed algorithms. Emek, Pfister, Seidel and Wattenhofer [13] proved that every problem that can be solved (and verified) by a randomized anonymous algorithm can also be solved by a deterministic anonymous algorithm provided that the latter is equipped with a distance-2 coloring of the input graph. Computing distance-k coloring is a key component in the derandomization of LOCAL distributed algorithms, due to Ghaffari, Harris and Kuhn [22] from 2019. Upper- and lower-bounds for approximate Minimum Dominating Set on power graphs were devised by Bar-Yehuda, Censor-Hillel, Maus, Pai and Pemmaraju [4] in 2020. Deterministic and randomized distance-2 coloring algorithms were obtained by Halldorsson, Kuhn and Maus [25]. Improved randomized results for distance-2 coloring were obtained by Halldorsson, Kuhn, Maus and Nolin [26]. Very recently, in 2023, Maus, Peltonen, and Uitto [33] devised deterministic algorithms for k-ruling sets on G^k with time $\tilde{O}(k^2 \log^4 n \log \Delta)$. They also devised randomized algorithm for this problem, as well as for MIS, with logarithm dependency on Δ and poly-log-log dependency on n. The most recent result for 2-distance coloring is a randomized algorithm by Flin, Halldorson and Nolin [15], whose running time is $O(\log^6 \log n)$.

2 Distance-2 coloring G with $O(\Delta^4)$ colors in $O(\log \Delta + \log^* n)$ rounds

In this section we devise an algorithm for distance-2 coloring of G using $O(\Delta^4)$ colors, which is a distance-1 coloring of G^2. Our algorithm significantly speeds-up the previously-known algorithms for distance-2 coloring with this number of colors. The previous algorithms [25] are based on simulating Linial's [30] algorithm in G^2. (See also [8], for more details about the original algorithm of Linial [30].) The algorithm of [25] for G^2 requires spending Δ rounds to simulate certain rounds of Linial's algorithm. On the other hand, our new method improves this, so that only $O(\log \Delta)$ rounds are required to simulate a round of Linial's algorithm. Consequently, our algorithm produces a proper $O(\Delta^4)$ coloring of G^2 within $O(\log \Delta + \log^* n)$ rounds in the CONGEST model. Moreover, most rounds of Linial's algorithm can be simulated within O(1) rounds.

The principle of our method is binary search. Consider the i-th round, $i = 1, 2, ..., O(\log^* n)$, of Linial's original algorithm. Each vertex v generates a subset of possible colors $S(\phi(v)) = \{s_1, s_2, ..., s_j\}$ from the palette of $\{1, 2, ...(\Delta \cdot \log^{(i)} n)^2\}$, such that there exists a color $s_j \in S(\phi(v))$, where $s_j \notin S(\phi(u))$, for all neighbours u of v. Moreover, Linial's algorithm makes it possible to construct a set system, such that for any pair of neighbours u,v, it holds that

$$\frac{|S(\phi(u))|}{|S(\phi(u)) \cap S(\phi(v))|} > \Delta \text{ and } \frac{|S(\phi(v))|}{|S(\phi(u)) \cap S(\phi(v))|} > \Delta.$$

Each vertex v selects such a color s_j, which is from its own set $S(\phi(v))$, but does not belong to any of its neighbours sets, from which the neighbours select colors. Consequently, the coloring is proper. Since in each round the subsets are taken from smaller sets, the number of colors is reduced in each round. We show that the element s_j can be found using binary search, without knowing the neighbours sets $S(\phi(u))$, but only knowing the number of intersections with neighbours' sets in a specific range. When solving 2-distance-coloring using this idea in a straightforward way, each node needs to receive all the subsets from 2-distant neighbours. This causes all nodes to send messages with size of at least $O(\Delta)$ for each node in order to receive messages with all of its distance-2 neighbours colors. (Each neighbor of a given vertex sends it the colors of all its own neighbours.) Using this information a vertex can compute the available color to choose. However, this approach exchanges much more information than needed and can be optimized for restricted bandwidth models. We provide this optimization in Section 2.1.

2.1 High level description

Our technique does not use an ordinary set, but an ordered set, thus we can perform a binary search. The goal of the binary search, for a vertex $v \in V$, is finding an element in $S(v)$, that does not belong to any set $S(u)$, for u in the 2-hop-neighborhood of v. To this end, for each vertex we define a range, that initially contains all elements of $S(v)$. Then we reduce the range size by a factor of at least 2 in each stage of the binary search. Eventually, each $v \in V$ reduces its range to contain a single element that does not belong to any range in its 2-hop-neighborhood. However, there is a cost in running time because each binary search requires $O(\log k)$ rounds where k is the size of the colors palette of the current stage.

The technique in high level is that each vertex knows its 1-distance neighbours subsets that are based on the coloring $\phi(u)$ and marked $S(\phi(u))$ and computes the number of intersections with these subsets. Each node holds two indices that constitute the beginning and end of the relevant range. The neighbours are aware of those indices and refer to the beginning index by *left* and the ending index by *right*. On every iteration the number of intersecting values of the subsets is reduced by a factor of at least two, simply by counting the elements in each half of a range and choosing the half-range with less intersections. On each round each vertex receives the number of intersecting values for both their left half and right half from its neighbours and decides whether in the next round it will use the left half of the range or the right half, and update its $left, right$ indices accordingly. The selected half of the range is the one with fewer intersecting values.

Next, we provide the pseudocode of our algorithm (see Algorithm 1 below), called *2-Distance-Linial*, and analyze its correctness and running time.

Algorithm 1 2-Distance-Linial's algorithm phase.

```
Let $t$ be the size of each set in the current phase /* All sets have the same size in the same phase */
Let $S(v) = S(\phi(v)) = \{s_1(v), s_2(v), ..., s_t(v)\}$ be the ordered set produced by the algorithm of Linial for the vertex $v$. /* $S(\phi(v))$ is computed locally by Linial's algorithm as a function of $\phi(v)$. */
Let $S(u_i) = S(\phi(u_i)) = \{s_1(u_i), s_2(u_i), ..., s_t(u_i)\}$
From Linial's proof we know that every $S$ contains an element $s_i$, such that $s_i$ belongs to $S$ and does not belong to any other $S(\phi(u_j))$ with $\phi(u_j) \neq \phi(u_i)$
The vertex v performs a binary search in the set S, with assistance of its neighbours, as follows:
$left = 1, right = t$
while $left \neq right$ do
    for $u \in \Gamma(v)$ in parallel do
        $Int_l(v,u)$ = Number of intersections of $S(v)$ and $\{s_i(u) \mid i \in [left, ..., \frac{right}{2} - 1]\}$
        $Int_r(v,u)$ = Number of intersections of $S(v)$ and $\{s_i(u) \mid i \in [\frac{right}{2}, right]\}$
    end for
    All neighbours $u$ of $v$ notify $v$, in parallel, about $Int_l(u,v)$ and $Int_r(u,v)$.
    All nodes in parallel send $Int_l(u_i,v)$ and $Int_r(u_i,v)$ to 1-hop-neighbours $u_i$
    A node $v$ receives $Int_l(u,v)$ and $Int_r(u,v)$ from all of its neighbours
    The vertex $v$ computes the sums $sum_l = \sum_{u \in \Gamma(v)} Int_l(u,v)$ and
    $sum_r = \sum_{u \in \Gamma(v)} Int_r(u,v)$.
    if $sum_r \geq sum_l$ then
        $right = \frac{right}{2} - 1$
        Send to all neighbours "left chosen"
    else
        $left = \frac{right}{2}$
        Send to all neighbours "right chosen"
    end if
    The new set $S(v)$ for the next phase of the binary search is
    $S(v) = \{s_{left}(v), ..., s_{right}(v)\}$
    Receive all neighbours [left \ right] choices and compute $S(u)$ for all $u \in \Gamma(v)$
end while
return the color $s_{left}(v)$        /* now $s_{left}(v) = s_{right}(v)$ */
```

2.2 Proof and run time analysis

▶ **Lemma 2.1.** *After each invocation of 2-Distance-Linial's algorithm phase the coloring ϕ remains proper.*

▶ **Lemma 2.2.** *The running time of 2-distance-Linial in the CONGEST model is* $O(\log^* n \cdot \log \Delta + \log \log n)$.

Details and proofs of the lemmas in section 2.2 appear in the full version of this paper [10].

Next, we provide an improvement, which removes the $O(\log \log n)$ factor from Lemma 2.2. To this end, we perform each binary search for $O(\log \Delta)$ phases, rather than $O(\log t)$. Moreover, in each phase we send just one bit to indicate whether the left half is chosen or the right one, rather than sending indices of ranges. This information is sufficient to compute the new range from the previous one. After $O(\log \Delta)$ phases, we obtain consecutive range of size $O(t/\Delta) = O(\log n)$. Recall that the previous range size is $t = O(\Delta \cdot \log n)$, which is a square root of the number of colors. The new range defines a bit-string of size $O(\log n)$ that represents whether there is a conflict for each element in $\{s_{left}(v), ..., s_{right}(v)\}$. This string is then sent directly to v by each of its 1-hop-neighbours. This is done using $O(\log n)$-bits messages, within one round in parallel by all neighbours. Then v finds an index $i \in O(\log n)$ of a bit with a 0 value, which exists since there is an index without conflicts in the range $\{s_{left}, ..., s_{right}\}$. The resulting color with no conflicts is $s_{left+i}(v)$. We summarize this in the next theorem.

▶ **Theorem 2.3.** *A proper distance-2 coloring with $O(\Delta^4)$-colors can be computed in $O(\log \Delta \cdot \log^* n)$ rounds in the CONGEST model.*

Theorem 2.3 demonstrates that for each of the $\log^* n$ iterations of Linial's algorithm, $O(\log \Delta)$ rounds are performed to compute the color for the next iteration. We now argue that it is sufficient to perform just two iterations with $O(\log \Delta)$ rounds, while the remaining $O(\log^* n)$ iterations require $O(1)$ rounds each. The idea is similar to an improvement from $O(\Delta \log^* n)$ to $O(\Delta + \log^* n)$ of [25]. Specifically, after 2 iterations the number of colors becomes $O(\Delta^4 \log^2 \log n)$. If $\Delta^4 > \log^2 \log n$. Then this is an $O(\Delta^8)$-coloring. It can be converted into an $O(\Delta^4)$-coloring within a single iteration, using a field of size $\Theta(\Delta^2)$, and polynomials of degree 4. Indeed, each of the current $O(\Delta^8)$ colors can be assigned a unique polynomial, and each such polynomial has at least one non-intersecting point with any Δ^2 others. These are used for computing new colors in a range of size $O(\Delta^4)$. The other possibility is that $\Delta^4 \leq \log^2 \log n$. Then, instead of performing a binary search, one can directly send Δ messages, each of which consists of a color in a range of $O(\Delta^4 \log^2 \log n) = O(\log^4 \log n)$. Since $\Delta \leq \sqrt{\log \log n}$, the number of bits in the concatenation of these Δ messages is poly$(\log \log n)$, so it can be sent over an edge within $O(1)$ rounds of the CONGEST model.

▶ **Corollary 2.4.** *A proper distance-2 coloring with $O(\Delta^4)$-colors can be computed in $O(\log \Delta + \log^* n)$ rounds in the CONGEST model.*

3 Distance-2 coloring with $O(\Delta^2)$ colors in $O(\Delta \cdot \log \Delta + \log^* n)$ rounds

3.1 High level description

In this section we provide an improved algorithm for distance-2 coloring of G using $O(\Delta^2)$ colors, which is distance-1 coloring of G^2. The improvement is from $O(\Delta^2 + \log^* n)$ rounds to $O(\Delta \cdot \log \Delta + \log^* n)$ rounds in the CONGEST model. The result is achieved by applying our technique to the algorithm of [9] that provides an $O(\Delta)$ coloring of an input graph G in $O(\sqrt{\Delta} \log \Delta + \log^* n)$ rounds. The algorithm of [9] is based on the following notions.

A *p-defective coloring* is a vertex coloring such that each vertex may have up to p neighbours with its color.

An *p-arbdefective coloring* is a vertex coloring, such that each subgraph induced by a color class of the coloring has arboricity bounded by p. The *arboricity* is the minimum number of forests into which the edge set of a graph can be decomposed.

The algorithm of [9] for distance-1 coloring consists of three stages:

1. Computing $O(\sqrt{\Delta})$-defective $O(\Delta)$-coloring of G in $O(\log^* n)$ time.
2. Computing $O(\sqrt{\Delta})$-arbdefective $O(\sqrt{\Delta})$-coloring of G in $O(\sqrt{\Delta})$ time.
3. Iterating over the $O(\sqrt{\Delta})$ color classes of step 2, and computing a proper coloring of G iteratively. Each iteration of Stage 3 requires a constant number of rounds.

As a first step of our extension of this scheme to work in G^2, we introduce the following *proxy* communication method. The goal is establishing a single path between any pair of vertices that need to communicate, and are at distance at most 2 one from another. (In section A.2, we generalize this to vertices of distance k one from another.) This way, the desired information is passed only once, which improves communication costs, and avoids miscalculations caused by duplicated data. To this end, for each vertex, a BFS tree of radius two that is rooted at the vertex is computed. This computation starts with sending the list of neighbours of each vertex to all its neighbours. This requires $O(\Delta)$ rounds, because in each round each vertex sends the information to a neighbor that has not been sent yet. Then, each vertex is aware of the neighbours of its neighbours. For any distance-2 neighbor, it knows the immediate neighbours that connect to it, and selects exactly one of these neighbours. The selected neighbor is referred to as *proxy*. Note that a vertex v with its proxy nodes and their neighbours form a BFS of radius two that contains the two-hop neighborhood of v.

In Sections 3.2.1 - 3.2.3, we describe the generalizations and modifications in the above-mentioned stages (1) - (3), for coloring G^2 in $O(\Delta \cdot \log \Delta + \log^* n)$ rounds in the CONGEST setting. For these computations we need the above-mentioned notion of proxy nodes.

The running time analysis assumes that the proxy nodes have already been computed. Otherwise, an $O(\Delta)$ term should be added. However, this does not affect the overall running time of the entire algorithm of Section 3, which is $O(\Delta \log \Delta + \log^* n)$.

3.2 Detailed description of the algorithm

3.2.1 Our variant for distance-2 defective coloring

The computation of $O(\Delta)$-defective $O(\Delta^2)$-coloring of G^2 proceeds as follows.
The procedure starts by computing a proper $O(\Delta^4)$-coloring of G^2, using Algorithm 1. Next, find a prime $q = \Theta(\Delta)$, such that the number of colors is bounded by q^4. Note that each color is represented by a tuple $\langle a, b\rangle, a, b \in q$ and Z_q is a field modulo q. Assign each color a unique polynomial $p(x) = a + b \cdot x + c \cdot x^2 + d \cdot x^3$, such that $0 \le a, b, c, d \le q - 1$. Assign each vertex a polynomial according to its color. We say that two polynomials $p(x), p'(x)$ *intersect* at the value t,$0 \le t \le q - 1$, if $p(t) = p'(t)$. Next, each vertex $v \in V$ finds a value $t, 0 \le t \le q-1$, such that $p(t)$ intersects with the minimum number of polynomials of vertices of distance at most 2 from v. This is done as follows, by a binary search. The vertex v sends its polynomial to its 1-hop neighbours. Each of these neighbours $u \in \Gamma(v)$ computes the number of intersections of v's polynomial with polynomials of neighbours w of u, such that u is the proxy for $\{v, w\}$. In addition, each $u \in \Gamma(v)$ computes the number of intersections of its polynomial with that of v. The number of intersections is computed for each half of the range $\{0, 1, ..., q-1\}$, i.e., $\{0, 1, ... \lceil q/2\rceil\}$ and $\{\lceil q/2\rceil + 1, \lceil q/2\rceil + 2, ...q - 1]\}$. This information is returned to v by all its 1-hop neighbours. Then v knows how many intersections with its 2-hop neighbours its polynomial has in $\{0, 1, ... \lceil q/2\rceil\}$ and $\{\lceil q/2\rceil + 1, \lceil q/2\rceil + 2, ...q - 1]\}$. The half-range with fewer intersections is selected for the next iteration of the binary search. This is repeated for $\log q$ iterations, until the range contains a single element $t \in \{0, 1, ..., q - 1\}$. The color of v returned by the procedure is $\langle t, p(t)\rangle$. This completes the description of the procedure. Its correctness and running time are analyzed below.

▶ **Lemma 3.1.** *The procedure computes an $O(\Delta)$-defective $O(\Delta^2)$-coloring of G^2.*

Proof's details appear in the full version of this paper [10].

▶ **Lemma 3.2.** *The running time of the procedure is $O(\log \Delta + \log^* n)$.*

Proof. First, in order to compute $O(\Delta^4)$ coloring we employ Algorithm 1 that has running time $O(\log\Delta+\log^* n)$, by Corollary 2.4. The remaining part of the procedure is a binary search on a range of size $O(\Delta^4)$, and thus requires $O(\log(\Delta^4)) = O(\log\Delta)$ phases, each of which consists of a constant number of rounds. The overall running time is $O(\log\Delta + \log^* n)$. ◀

3.2.2 Algorithm for Distance-2 Arbdefective Coloring

For a graph G, given an $O(\Delta)$-defective $O(\Delta^2)$-coloring of G^2, we would like to produce an $O(\Delta)$-arbdefective $O(\Delta)$-coloring of G^2 within $O(\Delta)$ rounds. The algorithm is as follows:

This algorithm extends the ideas of [9] to work in G^2 in the CONGEST model. In that paper the authors devised an $O(\sqrt{\Delta})$-arbdefective $O(\sqrt{\Delta})$-coloring algorithm for G with $O(\sqrt{\Delta} + \log^* n)$ rounds. The main idea of the algorithm is as follows. In each round, each vertex counts how many of its neighbours have the same color as its own. The number of such neighbours is the *number of conflicts.* If a vertex has too many conflicts, it selects a new color, using a certain function. Otherwise, the vertex finalizes its color. The original algorithm [9] proceeds for $O(\sqrt{\Delta})$ rounds, and selects the round with the smallest number of conflicts. By the pigeonhole principle, there must be a round in which the number of color conflicts is at most $O(\sqrt{\Delta})$. However, computing the number of conflicts with all 2-hop-neighbours is expensive, since each original round can take up to $O(\Delta)$ rounds, when applied in a straightforward way to G^2. To improve this, each vertex collects information about conflicts from its 2-hop-neighborhood in a bit-efficient manner. Specifically, a vertex receives from each of its 1-hop neighbours the number of conflicts it has with 2-hop-neighbours, instead of lists of their colors. During the execution of the algorithm, in each iteration the conflicts are counted, and if the total number of 2-distance conflicts is below a predefined t, the vertex finalizes the current color. For any $t \in [1, \Delta^2]$, this stage requires $O(\frac{\Delta^2}{t})$ time and it results in $O(t)$-arbdefective $O(\frac{\Delta^2}{t})$-coloring of G^2. Setting $t = O(\Delta)$ results in $O(\Delta)$-arbdefective $O(\Delta)$-coloring in time $O(\Delta)$. See pseudocode of Algorithms 2 - 3 below. (Each color in an initial $O(\Delta^2)$-coloring is represented by an ordered pair $\langle a, b\rangle$, where $a, b \in O(\Delta)$. When Algorithm 3 terminates, the resulting color resides in the b-coordinate, and it is in the range $[0, 1, ..., O(\Delta)]$.) Next, we analyze Algorithm 3.

Algorithm 2 Procedure number-of-conflicts $(\langle a_0, b_0\rangle,\ \langle a_1, b_1\rangle)\ ..\ \langle a_n, b_n\rangle$).

```
/* This procedure is performed internally by a vertex, within 0 rounds */
numberOfConflicts = 0
for $i = 1, 2, ..., n$ do
    if $b_0 = b_i$ then
        numberOfConflicts = numberOfConflicts + 1
    end if
end for
return numberOfConflicts
```

Algorithm 3 2-Distance AG Arbdefective Coloring($maxDefect$ = t).

```
 1: We are given a $p$-coloring $\phi$. Denote $q$ as the smallest prime number such that $q \geq \sqrt{p}$.
    The parameter $maxDefect$ is the maximal arb-defect allowed for coloring.
 2: Denote $\phi(v) = \langle a, b \rangle$, where $a, b \leq q$
 3: while $\phi(v) \neq \langle 0, b \rangle$ do
 4:     Denote by $conflicts(v, u) \leftarrow$ number-of-conflicts($\phi(v) \bigcup_{u_i \in \Gamma(u)} \phi(u_i)$)
 5:     for $i = 1, 2, .., deg(v)$ in parallel do
 6:         send to the $i$th neighbor of $v$, which is $u_i$, the message $conflicts(u_i, v)$
 7:     end for
 8:     Receive all $conflicts(v, u_i)$ messages from neighbours
 9:     if $\sum_{(v,u_i) \in E} conflicts(v, u_i) \leq maxDefect$ then
10:         $\phi(v) = \langle 0, b \rangle$
11:         Send "Done" to all neighbours
12:     else
13:         $\phi(v) = \langle a, a + b \bmod q \rangle$
14:         Send "Not done" to all neighbours
15:     end if
16:     Receive all "Done", "Not Done" messages from neighbours, and compute $\phi(u_i)$ for
        $i = 1, 2, ..., deg(v)$
17: end while
```

▶ **Lemma 3.3.** *After running 2-Distance AG Arbdefective Coloring for $\lceil 2\Delta^2/maxDefect \rceil$ rounds, all vertices have colors of the form $\langle 0, b \rangle$, and each color class has arboricity at most $maxDefect$ in G^2.*

Proof's details appear in the full version of this paper [10].

3.2.3 Iterative Algorithm for Distance-2 Proper Coloring

In this subsection we describe an algorithm that produces an $O(\Delta^2)$-proper-coloring of G^2 within $O(\Delta \cdot \log \Delta + \log^* n)$ rounds. This algorithm is based on a the technique devised in [9], but our algorithm extends this technique to work for G^2. For more details regarding this technique, see section 3 in [5]. Our new algorithm starts with computing an $O(\Delta)$-arbdefective $O(\Delta)$-coloring φ for G^2. The coloring φ constitutes a partition of the graph into $O(\Delta)$ color sets $V_1, V_2, ..., V_d$, $d \in O(\Delta)$. Each color class is $O(\Delta)$-arbdefective. This means that the arboricity of a subgraph of G^2 induced by V_j, $j \in O(\Delta)$, is bounded by $O(\Delta)$. Moreover, each pair of vertices u, v at distance at most 2 one from another have a parent-child relation in a certain forest. Specifically, when an arbdefective coloring is computed with Algorithm 3, if u, v terminate (arrive to step 10 of the algorithm) in distinct rounds, then the parent is the vertex that terminated earlier. Otherwise, the parent is the vertex with lower ID. Vertices do not have to know their parents explicitly.

The algorithm iterates over $i = 1, 2, ..., d$. In each iteration i the algorithm computes a new color φ' for all of the nodes with color $\phi(v) = i$, using at most $O(\Delta^2)$ colors. To this end, each vertex constructs a set of polynomials, and finds a polynomial P in this set, such that:

(1) The number of intersections of the polynomial P with colors φ' of vertices in its 2-hop neighborhood that already selected such colors in previous rounds is as small as possible.

(2) The number of intersections of P with polynomials of its parents in its 2-hop-neighborhood that are active in the same round i is as small as possible.

The construction of the polynomial set of a vertex $u \in V_i$ is performed as follows. Let $q = O(\Delta)$ be a prime, such that $q > c \cdot \Delta$, for a sufficiently large constant $c \geq 1$. We represent the color $\varphi(u)$ by $\langle a, b \rangle$, where $0 \leq a, b < q$. The set of polynomials of u is $\{a \cdot x^2 + b \cdot x + j \mid j = 0, 1, ..., q-1\}$. The number of polynomials in the set is $q = O(\Delta)$.

According to (1), our goal is finding a polynomial P in the set of u, such that the number of vertices at distance at most 2 from u with the following property is minimized.

(*) For a vertex w that already has a color $\varphi'(w) = \langle a'_w, b'_w \rangle$, there exists $t \in 0, 1, ..., q-1$, such that $\langle a'_w, b'_w \rangle = \langle t, P(t) \rangle$.

According to the Pigeonhole principle, there must be a polynomial in the set of size $q > c \cdot \Delta$, for which at most $q/2$ vertices at distance at most 2 satisfy this property. This is because each vertex satisfies this property for at most one polynomial in this set (the set consists of non-intersecting polynomials), and the number of vertices at distance at most 2 is at most $q^2/2$. Our goal is finding such a polynomial. The challenge is that when running a naive version of this algorithm in the CONGEST model, every vertex needs to know its 2-hop neighbours' colors. Sending this information requires $O(\Delta)$ rounds. Next we describe an optimization that requires only to compute how many intersections there are in sets of polynomials. This speeds up the running time from $O(\Delta)$ to $O(\log \Delta)$.

Next, we describe how each vertex selects the desired polynomial $a \cdot x^2 + b \cdot x + j$ from its set within $O(\log \Delta)$ rounds. This is done using a binary search on j. To this end, each vertex has to inform its neighbours about its set of polynomials. Even though there are q polynomials in the set, this is done just within one round, as follows. Given a set of polynomials $\{a \cdot x^2 + b \cdot x + j \mid j = 0, 1, ..., q-1\}$ of a vertex u, only the coefficients a, b are sent to the neighbours of u. (Each coefficient requires $O(\log \Delta)$ bits.) Then the neighbours can reconstruct the set of polynomials from a, b, since they know that j runs from 0 to $q-1$. Next, every vertex initialize $start = 1$ and $end = q$ and defines two ranges. The ranges are $low = [start, \lceil \frac{end - start}{2} \rceil]$ and $high = [\lceil \frac{end - start}{2} \rceil + 1, end]$. At the first step each vertex sends to each of its neighbours w the number of intersections of colors φ' with polynomials that have j in range low, as well as the number of intersections for j in range $high$. In the next step each vertex receives from its neighbours the number of such intersections in ranges low and $high$ in its 2-hop-neighborhood. Then each vertex decides for its new $start$ and end according to the half range in which there are fewer intersection with its polynomials. Consequently, after halving $O(\log \Delta)$ times, the range contains just a single value $\hat{j}$. It defines a single polynomial from the set, which is $a \cdot x^2 + b \cdot x + \hat{j}$.

The next lemma provides a helpful property of the polynomials, which will assist us to compute the coloring of a set V_i, given colorings of $V_1, V_2, ..., V_{i-1}$.

▶ **Lemma 3.4.** *Suppose that we are given a graph with $O(\Delta)$-arbdefective $O(\Delta)$-coloring φ of G^2, that partitions the input graph into subsets $V_1, V_2, ..., V_{O(\Delta)}$, according to color classes of φ. Moreover for an integer $i \geq 0$, suppose that we already have a proper 2-distance coloring φ' for $V_1, V_2, ..., V_{i-1}$. Then we can find a polynomial $P = a \cdot x^2 + b \cdot x + \hat{j}$ for each vertex u with $\varphi(u) = i$, such that at least half of the elements in the set $\{\langle 0, P(0) \rangle, \langle 1, P(1) \rangle ..., \langle q-1, P(q-1) \rangle\}$ does not appear as φ' colors in the 2-hop neighborhood of u.*

Proof's details appear in the full version of this paper [10].

According to Lemma 3.4, it is possible to iterate over the color classes of the arbdefective coloring φ, for $i = 1, 2, ..., O(\Delta)$. In each iteration i, each vertex in the color class i obtains a single polynomial with the properties stated in the lemma. Specifically, it has sufficiently many elements that still can be used for their φ' color. Specifically, the number of elements is larger (by a factor greater than 2) than the number of their parents in G^2. Consequently, a variant of Linial's algorithm that considers only parents in the 2-hop-neighborhood can be executed. (For the case of distance-1 coloring, this is a well-known extension of Linial's algorithm, which is called arb-Linial [6].) In the case of distance-2 coloring it can be computed in $O(\log \Delta)$ phases in the same way as in Corollary 2.4, but considering only 2-hop-parents, rather than entire 2-hop-neighborhood. Recall that a vertex can deduce the parent-child relationship of a pair of its neighbours, by inspecting their termination round in Algorithm 3 and their IDs. Hence, we obtain the following Corollary.

▶ **Corollary 3.5.** *It is possible to compute a proper distance-2 coloring with $O(\Delta^2)$ colors within $O(\Delta \cdot \log \Delta + \log^* n)$ rounds in the CONGEST model.*

3.2.4 Coloring G^2 using $(\Delta^2 + 1)$ colors in $O(\Delta^{\frac{3}{2}} \cdot \log \Delta + \log^* n)$ rounds

In this section we show how to reduce the number of colors to $(\Delta^2 + 1)$. To this end, we parameterize the steps of our scheme in a different way, as follows.

1. Compute an 0-defective $O(\Delta^2)$-coloring of G^2, i.e., a proper $O(\Delta^2)$-coloring.
2. Compute $O(\sqrt{\Delta})$-arbdefective $O(\Delta^{3/2})$-coloring of G^2.
3. Iterate over the $O(\Delta^{3/2})$ color classes that were generated in step 2, and compute a proper coloring of G^2 iteratively, using $\Delta^2 + O(\Delta^{3/2})$ colors.
4. Apply a simple reduction to produce a (Δ^2+1)coloring of G^2 from a $\Delta^2+O(\Delta^{3/2})$-coloring.

The steps are performed as follows. Step 1 is obtained by applying Corollary 3.5. This step requires $O(\Delta \cdot \log \Delta + \log^* n)$ rounds. Step 2 is obtained by applying Lemma 3.3 with $maxDefect = \sqrt{\Delta}$ and $q = \Theta(\Delta^{3/2})$. This step requires $O(\Delta^{3/2})$ rounds. Step 3 is performed similarly to Section 3.2.3, but now we have $O(\Delta^{3/2})$ color classes to iterate on, rather than $O(\Delta)$. On the other hand, the arboricity of each of them is significantly smaller, and consequently a proper coloring from a range of size $\Delta^2 + O(\Delta^{3/2})$ can be computed. To this end, let q be a prime, such that $q > \Delta + O(\sqrt{\Delta})$, $q < 2 \cdot \Delta + O(\sqrt{\Delta})$. Then each vertex v in G^2 has at most Δ^2 neighbours which have finalized their colors in previous rounds, thus it cannot select their colors. By the pigeonhole principle, there exists a polynomial P of v, such that P intersects with at most $\frac{\Delta^2}{q}$ neighboring colors. This polynomial is defined over a field of size q, and there is a value $t \leq \Delta^2/q + O(\sqrt{\Delta})$, such that $\langle t, P(t) \rangle$ does not intersect with any neighboring polynomial, nor with any neighboring color. Thus, $\langle t, P(t) \rangle$ is selected as the color of v, and it is from a range of size $q \cdot (\frac{\Delta^2}{q} + O(\sqrt{\Delta})) + q = \Delta^2 + O(\Delta^{3/2})$. This is the resulting number of colors of step 3, whose running time is $O(\Delta^{3/2} \cdot \log \Delta)$ as described in Section 3.2.3.

Next we describe step 4, which reduces the number of colors from $\Delta^2 + O(\Delta^{3/2})$ to $\Delta^2 + 1$. This step is performed using an adaptation of a simple color reduction for G to G^2. The simple color reduction for G works as follows. Each vertex whose color is greater than all its neighbours colors (and greater than $\Delta(G) + 1$), selects a new color from $\{1, 2, ..., \Delta(G) + 1\}$ that is not used by any neighbor. By starting from a proper $(\Delta(G) + k)$-coloring, for a parameter $k > 1$, and repeating this for $k - 1$ rounds, a proper $(\Delta(G) + 1)$-coloring is achieved. If message size is unbounded, this can be directly applied in G^2. However, each vertex needs to collect the colors of its 2-distance neighbourhood. This is in order to know which colors are available. Thus, $O(\Delta)$ rounds are required to simulate each original round. To perform this in the CONGEST setting efficiently, we improve this by invoking a binary search to find an available color. This is done as in Section 2 and requires $O(\log \Delta)$ rounds. Consequently, an $O(\Delta^2 + O(\Delta^{3/2}))$-coloring of G^2 can be reduced into $(\Delta^2 + 1)$-coloring of G^2 within $O(\Delta^{3/2} \log \Delta)$ rounds. This completes the description of step 4. The result of invoking steps 1 to 4 is summarized in the next corollary.

▶ **Corollary 3.6.** *$(\Delta^2 + 1)$-coloring of G^2 can be computed within $O(\Delta^{3/2} \log \Delta + \log^* n)$ rounds in the CONGEST setting.*

4 Speedup technique for algorithms on G^k in the CONGEST model

In this section we devise a method which speeds up the running time of a wide class of algorithms for problems on G^k in the CONGEST model. This includes algorithms for problems such as: MIS, maximal matching, edge coloring, vertex coloring, ruling set, cluster decomposition, etc. Our goal is reducing the amount of data passed in the network, by exchanging messages only half-the-way, compared to standard algorithm for G^k. In the general case, in a problem for k-distance, a node may have up to Δ^k neighbours. Thus, previously-known solutions require Δ^{k-1} rounds for collecting information about k-hop neighbours in each step of an algorithm. This way, an algorithm with $f(\Delta, n)$ rounds for G is translated into an algorithm with $f(\Delta^k, n) \cdot \Delta^{k-1}$ rounds for G^k. (In general, the running time may be even larger, but we focus on solutions in which each vertex makes a decision based on the current information of vertices in its k-hop-neighborhood, where each vertex holds $O(\log n)$ bits.) On the other hand, our new technique makes it possible to collect aggregated data from distance only $k/2$. As a result, the size of the collected data becomes $O(\Delta^{k/2})$. This allows us to obtain a running time of $f(\Delta^k, n) \cdot \Delta^{k/2-1}$ instead of $f(\Delta^k, n) \cdot \Delta^{k-1}$. Nowadays, various problems have algorithms with running time $f(\Delta, n) = O(\text{polylog}(\Delta) + \log^* n)$ for G. In such cases our technique provides a quadratic improvement for G^k, i.e., for distance-k computations on G. Our technique is based on idempotent functions, described below.

4.1 Idempotent functions

An *idempotent function* is a function f from a set A to itself, such that for every $x \in A$, it holds that $f(f(x)) = f(x)$. For example, the following boolean functions from $\{0, 1\}$ to $\{0, 1\}$ are idempotent. $F_0(x) = x$ OR 0 , $F_1(x) = x$ OR 1 , $G_0(x) = x$ AND 0 , $G_1(x) = x$ AND 1. Another example, are functions $\hat{H}_t, \check{H}_t : N \to N$, where $t \in N$, defined as follows. $\hat{H}_t(x) = max(x, t)$, $\check{H}_t(x) = min(x, t)$.

A pair of functions $f(), g()$ is *commutative* if $f(g(x)) = g(f(x))$. A set of functions is a *commutative set* if any pair of functions in the set is commutative.
We define an *idempotent composition* as follows. A set A with a commutative set of functions $f_1, f_2, ..., f_k$ from A to itself is an *idempotent composition*, if for any q (not necessarily distinct) indices $j_1, j_2, ..., j_q$, in the range $[k]$, and $p \leq q$ distinct indices $i_1 \neq i_2, ..., \neq i_p$, such that $\{i_1, i_2, ..., i_p\} = \{j_1, j_2, ..., j_q\}$, it holds that

$$f_{j_1}(f_{j_2}(...f_{j_q}(x))...)) = f_{i_1}(f_{i_2}(...f_{i_p}(x))...)).$$

For example, the set $\{0, 1\}$ with the functions F_0, F_1 as defined above is an idempotent composition. Indeed, $F_0(F_0(...F_0(x)...)) = F_0(x) = x$, $F_1(F_1(...F_1(x)...)) = F_1(x) = 1$, and any composition of functions F_0 and F_1 equals $F_0(F_1(x)) = F_1(F_0(x)) = 1$.

4.2 High level description of our technique for G^k

This section assumes k is even. In the case k is odd our technique cost another factor of $O(\Delta)$ of communication rounds per algorithm round. Hence it behaves as if the distance required is $k + 1$ in terms for CONGEST communication. Our method for distance-k computations consists of two stages.

In the first stage each vertex collects information from its $k/2$-hop-neighborhood. This is done by broadcasting in parallel from all vertices to distance $k/2$. Consequently, for any pair of vertices u, v at distance k one from another, there exists a vertex w in the middle of a path between u and w that received the information of u and v. Indeed, the distance between u and w is $k/2$, and between v and w it is $k/2$. Next, w computes internally, for each u in the $k/2$-hop-neighborhood of w, the available information for u regarding its k-hop-neighbours. That is, the neighbours at distance k from u who are also in the $k/2$-neighborhood of w, Which, sometimes contains all of the vertices in the $k/2$-neighborhood of w and sometimes only part of in case we would like to avoid double counting.

In the second stage, for all $v \in V$, all information computed for v in its $k/2$-hop-neighborhood should be delivered to v. Recall that the information computed by vertices in the $k/2$-hop-neighborhood of v is about the k-hop-neighborhood of v. But delivering the entire information in a straightforward way to v requires up to Δ^k rounds. Indeed, vertices at distance $k/2$ from v hold information of size up to $\Delta^{k/2}$, and the number of such vertices is up to $\Delta^{k/2}$. In order to reduce the amount of information that has to be passed, an aggregation function is used. Specifically, each vertex v collects information in a convergecast manner. That is, each vertex at distance $k/2$ from v sends information of size $\Delta^{k/2}$ to its neighbours. This requires $O(\Delta^{k/2})$ rounds. Now vertices at distance $k/2 - 1$ from v have received information from their neighbours at distance $k/2$ from v. But instead of sending all this information to v they perform an aggregation in which the information size shrinks to $\Delta^{k/2-1}$. In the following phase, the information size shrinks to $\Delta^{k/2-2}$, etc. See Figure 1.

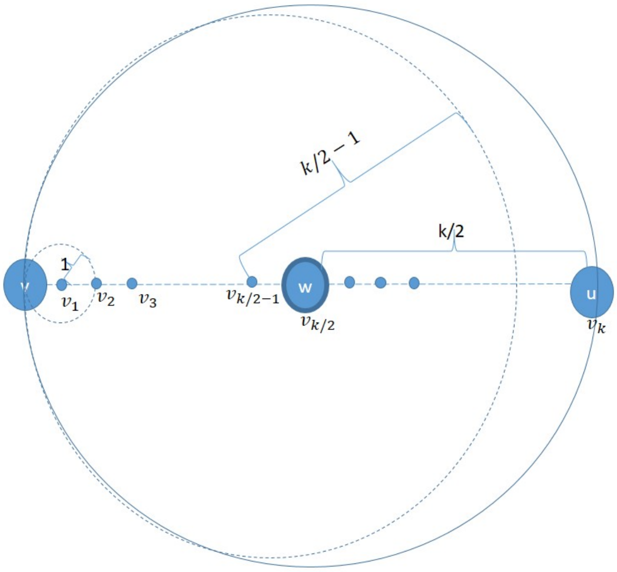


Figure 1 A vertex w in the middle of a path between v and u collects information about its $k/2$-hop-neighborhood. Then a convergecast process is performed, in which balls of radius $k/2 - i$, $i = 1, 2, ..., k/2$, around $v_{k/2-i}$ are formed within $k/2$ stages. The balls contain aggregated information that after $k/2$ stages is about the k-hop-neighborhood of v. After stage $k/2$ this information resides in a ball of radius 0 of v, i.e., in v itself.

Further details of our technique are available in the appendix below.

References

1 N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7(4):567–583, 1986. doi:10.1016/0196-6774(86)90019-2.

2 B. Awerbuch, A. Goldberg, V. Luby, and M. Plotkin. Network decomposition and locality in distributed computation. in *proc. of the 30th annual symposium on foundations of computer science (focs). pp.*, pages 364–369, 1989.

3 A. Balliu, S. Brandt, J. Hirvonen, D. Olivetti, M. Rabie, and J. Suomela. Lower bounds for maximal matchings and maximal independent sets. poceeding of the 60th annual symposium on foundations of computer science (focs). *pp.*, pages 481–497, 2019.

4 R. Bar-Yehuda, K. Censor-Hillel, Y. Maus, S. Pai, and S. V. Pemmaraju. Distributed approximation on power graphs. In *Proceedings of the 39th Symposium on Principles of Distributed Computing (PODC)*, pages 501–510, 2020.

5 L. Barenboim. Deterministic $(\delta+1)$-coloring in sublinear (in δ) time in static, dynamic and faulty networks. *Journal of the ACM*, 63(5), 2016. doi:10.1145/2979675.

6 L. Barenboim and M. Elkin. Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition. In *Proc. of the 27th ACM Symp. on Principles of Distributed Computing*, pages 25–34, 2008.

7 L. Barenboim and M. Elkin. Distributed $(\Delta+1)$- coloring in linear (in Δ) time. In *Proc. of the 41st ACM Symp. on Theory of Computing*, pages 111–120, 2009.

8 L. Barenboim and M. Elkin. *Distributed graph coloring: Fundamentals and recent developments*. Springer Nature, 2022.

9 L. Barenboim, M. Elkin, and U. Goldenberg. Locally-iterative distributed $(\delta+1)$-coloring below szegedy-vishwanathan barrier, and applications to self-stabilization and to restricted-bandwidth models. In *Proceedings of the ACM Sym-posium on Principles of Distributed Computing (PODC)*, pages 437–446, 2018.

10 L. Barenboim and U. Goldenberg. Speedup of distributed algorithms for power graphs in the congest model. arXiv preprint, 2023. arXiv:2305.04358.

11 Y. Chang, W. Li, and S. Pettie. Distributed $(\delta+1)$-coloring via ultrafast graph shattering. *SIAM Journal on Computing*, 49(3):497–539, 2020. doi:10.1137/19M1249527.

12 R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.

13 Y. Emek, C. Pfister, J. Seidel, and R. Wattenhofer. Anonymous networks: randomization= 2-hop coloring. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, 2014.

14 S. Faour, M. Ghaffari, C. Grunau, F. Kuhn, and V. Rozhoň. Local distributed rounding: Generalized to mis, matching, set cover, and beyond. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). Society for Industrial and Applied Mathematics*, 2023.

15 M. Flin, M. Halldórsson, and A. Nolin. Fast coloring despite congested relays. In *37th International Symposium on Distributed Computing (DISC)*, 2023.

16 P. Fraigniaud, M. M. Halldorsson, and A. Nolin. Distributed testing of distance-k colorings. *Proc.*, 27th Coll. on Structural Information and Communication Complexity (SIROCCO), 2020.

17 P. Fraigniaud, M. Heinrich, and A. Kosowski. Local conflict coloring. In *Proceeding of the 57th annual symposium on foundations of computer science (FOCS)*, pages 625–634, 2016.

18 M. Fuchs and F. Kuhn. List defective colorings: Distributed algorithms and applications. In *37th International Symposium on Distributed Computing (DISC)*, 2023.

19 M. Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the twenty-seventh annual ACM-SIAM Symposium on Discrete Algorithms. (SODA)*, pages 270–277, 2016.

20 M. Ghaffari and C. Grunau. Faster deterministic distributed mis and approximate matching. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*. 2023, 2023.

21 M. Ghaffari, C. Grunau, and V. Rozhoň. Improved deterministic network decomposition. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA) . Society for Industrial and Applied Mathematics*, pages 2904–2923, 2021.

22 M. Ghaffari, D. G. Harris, and F. Kuhn. On derandomizing local distributed algorithms. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS). IEEE*, 2018.

23 M. Ghaffari and F. Kuhn. Deterministic distributed vertex coloring: Simpler, faster, and without network decomposition. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS). IEEE, 2022*, pages 1009–1020, 2022.

24 A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM Journal on Discrete Mathematics*, 1(4):434–446, 1988. doi:10.1137/0401044.

25 M. M. Halldorson, F. Kuhn, and Y. Maus. Distance-2 coloring in the congest model. in *proc. of the 39th acm symposium on principles of distributed computing (podc). pp.*, pages 233–242, 2020.

26 M. M. Halldorsson, F. Kuhn, Y. Maus, and A. Nolin. Coloring fast without learning your neighbours' colors. *Proc. of the 34th International Symposium on Distributed Computing (DISC)*, 39(1-39):17, 2020.

27 D. Harris, J. Schneider, and H. Su. Distributed (delta + 1)-coloring in sublogarithmic rounds. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing (STOC)*, pages 465–478, 2016.

28 K. Kothapalli, C. Scheideler, M. Onus, and C. Schindelhauer. Distributed coloring in $\tilde{O}(\sqrt{\log n})$ bit rounds. in proc,. *of the*, 20th International Parallel and Distributed Processing Symposium, 2006.

29 F. Kuhn and R. Wattenhofer. On the complexity of distributed graph coloring. in proc. 25th acm symp. *Principles of Distributed Computing, pp.*, pages 7–15, 2006.

30 N. Linial. Locality in distributed graph algorithms siam j. comput., 21 (1992). *pp.*, 193–201:275–290, 1992.

31 M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. on Computing*, 15:1036–1053, 1986. doi:10.1137/0215074.

32 Y. Maus. Distributed graph coloring made easy. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 362–372, 2021.

33 Y. Maus, S. Peltonen, and J. Uitto. Distributed symmetry breaking on power graphs via sparsification. In *Proceeding of the 42nd Symposium on Principles of Distributed Computing (PODC)*, pages 157–167, 2023.

34 A. Panconesi and A. Srinivasav. On the complexity of distributed network decomposition. *Journal of Algorithms*, 20(2):356–374, 1996. doi:10.1006/JAGM.1996.0017.

35 V. Rozhoň and M. Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 350–363, 2020.

36 M. Szegedy and S. Vishwanathan. Locality based graph coloring. in proc. In *25th ACM Symposium on Theory of Computing*, pages 201–207, 1993.

A Appendix

A basic building block of ours for the first stage is a procedure for broadcasting to distance $k/2$, initiated by all vertices in parallel. In this procedure, each message contains three elements: Node id of the originator of broadcast, the distance from the originator, and additional information. The part that contains the id and the distance is called *header*. In addition, each vertex v maintains a local data structure D_v that initially contains only the element of v. In the beginning of the broadcast procedure, each vertex sends a message with the contents of D_v (in the additional information field of the message) to all its neighbours. During the execution of the broadcast procedure, if a node receives several messages with the same id of the originator, the node considers only one of these messages (which was the first to arrive; ties are broken arbitrarily) and ignores the others. In each round of the broadcast, each vertex updates its data structure D_v with additional information it learnt, and sends it to its neighbours. Specifically the update of D_v considers each received message and each element in the message with ID that the vertex has not received yet. The vertex v adds each such element to D_v. Then the updated data structure D_v is sent to v's neighbours. The broadcast procedure continues this way until the distance reaches $k/2$. Each message with distance greater than $k/2$ from the origin is discarded. Thus the broadcast process stops after $k/2$ rounds from initiation.

Once the broadcast stops, for any vertex w at distance at most $k/2$ from v, the vertex w computes locally an appropriate value for v, using aggregation functions. These functions are applied iteratively for each vertex in the $k/2$-hop-neighborhood of w. Next, a convergecast process is performed for $k/2$ phases. Its goal is that each $v \in V$ learns the outcome of a series of function invocations that are applied to all vertices in the k-hop-neighborhood of v. (This is done with assistance of vertices at distance up to $k/2$ from v.) To this end, aggregation functions $f : A \to A, A \subseteq N_0$ are used. In certain cases A is of size 2, as in the example of deciding whether the value of v appears in its k-hop-neighborhood. But in other cases $|A|$ may be larger. It has to satisfy $\log(|A|) = o(\Delta^{k/2})$, to provide a speed-up in our method. The number of aggregation functions is $|A|$, one per element of A. (This is because each element may affect uniquely the aggregated data, and an appropriate function for each element is needed.) In each phase $i = 1, 2, ..., k/2$ of the convergecast process each vertex $v \in V$ receives from its neighbours the results of the aggregation functions applied to each vertex z in the $(k/2 + 2 - i)$-hop-neighborhood of v. (Note that these function applications hold results about distance $(k/2 + i)$ from v at that stage.) For each such vertex z, there are up to Δ results of functions application, one per neighbor of v. Then for each z' in the $(k/2 + 1 - i)$-hop-neighborhood of v, the vertex v applies the aggregation function iteratively, up to Δ times, according to the number of results for z' that v received from its neighbours. This shrinks the amount of information by a factor of Δ for the next phase, where the radius of function application becomes smaller by 1, and the radius of gained information becomes larger by 1. After $k/2$ phases, each vertex $v \in V$ holds a single result of the applications of aggregation functions on all vertices in the k-hop-neighborhood of v.

A key property of the above process, which constitutes its correctness, is provided in the next lemma.

▶ **Lemma A.1.** *After each phase $i = 1, 2, ..., k/2$, for each pair of vertices v, u in G at distance k one from another, there exists a vertex z at distance $k/2 - i$ from v whose result of applications of the aggregation functions until this stage includes an invocation for u.*

Proof's details appear in full version of the paper [10].

The above lemma shows that after i phases of the convergecast stage the $(k/2 - i)$-hop-neighborhood of each vertex v contains all needed information for v about its k-hop neighborhood. Indeed for any vertex u of distance up to k from v there is a vertex at distance $(k/2 - i)$ from v who holds a result of a series of applications of the aggregation functions, one of which is an invocation for v on u. Thus after $k/2$ phases each vertex $v \in V$ holds the result of invocations of the aggregation functions for all vertices in its k-hop-neighborhood. The number of rounds of each phase $i = 1, 2, ..., k/2$ is $O(\log |A| \cdot \Delta^{k/2+1-i})$.

We illustrate this scheme by devising an algorithm for computing k-distance $O(\Delta^k)$-coloring. The algorithm generalizes 2-distance algorithms as follows and based on the additive group coloring (AG) algorithm. The aggregation functions are the OR function, where $A = \{0,1\}$. Specifically: $F_0(x) = x$ OR 0 ; $F_1(x) = x$ OR 1. Each vertex $w \in V$ collects the current colors of its $k/2$-hop-neighborhood. For each vertex v in the $k/2$-hop-neighborhood of w, the vertex w computes whether the b (We indicate a vertex color by the notation $\langle a,b \rangle$ when we compute additive group coloring) element of the color of v equals to at least one of the b elements of the other vertices in the $k/2$-hop-neighborhood of w. To this end, for each vertex u in its $k/2$-hop-neighborhood, w applies a function $F_i(x)$, where $i = 1$ if the b values of v equals to that of u, and $i = 0$ otherwise. Each invocation of $F_i(x)$ by w, except the first invocation of a phase, is applied on the result of another invocation by w in that phase. If this is the first invocation, then it is applied on $x = 0$. Once the vertices apply the function for their $k/2$-hop-neighbours, the resulting $\Delta^{k/2}$ values are sent to immediate neighbours. Consequently, each vertex $v \in V$ receives information of the $k/2$-hop-neighborhoods of its own neighbours. Note that vertices at distance up to $k/2-1$ from v received information about vertices at distance up to k from v. Now the same is repeated, but for distance $k/2-1$, rather than $k/2$. This continues for $k/2$ phases. Finally, as a result of the function invocations and exchange of messages, each vertex $v \in V$ holds a single result, indicating whether there is another vertex in its k-hop-neighborhood with the same b value. This allows to compute the next color of v, as in the AG algorithm. Thus, within $O(\Delta(G^k)) = O(\Delta^k)$ iteration of computing the next colors as described above, a proper distance-k coloring with $O(\Delta^k)$ colors is obtained, from an initial $O(\Delta^{2k})$-coloring to distance k. The latter coloring can be obtained by generalizing 2-distance-Linial's algorithm (Algorithm 1) to distance k. (We elaborate on this in Section A.1.) Each iteration requires $O(\Delta^{k/2})$ rounds, and the overall running time is $O(\Delta^{k+k/2} + k \cdot \Delta^{k/2} \cdot \log\Delta \cdot \log^* n)$. We summarize this discussion with the next theorem.

▶ **Theorem A.2.** *Distance-k coloring with $O(\Delta^k)$ colors can be computed within $O(\Delta^{k+k/2} + k \cdot \Delta^{k/2} \cdot \log\Delta \cdot \log^* n)$ rounds in the CONGEST model.*

In Section A.1 we show how the running time for distance-k coloring can be improved.

We conclude the current section by improving the running time of simulating a round of an algorithm for G in G^k. The improvement is from $O(\Delta^{k/2})$ to $O(\Delta^{k/2-1})$. Consider the last phase of broadcast, when each vertex needs to receive information about $(k/2-1)$-neighborhoods from each of its immediate neighbours. A $(k/2-1)$-neighborhood contains up to $\Delta^{k/2-1}$ vertices, each of which holds $O(\log n)$ bits of data for transmission. Thus, $O(\Delta^{k/2-1})$ rounds are needed in the last phase of broadcast to deliver the required information to each vertex, which then employs it to compute locally the information regarding its $k/2$-hop-neighborhood. Then, in the first phase of convergecast, instead of sending the computation of the entire $k/2$-hop-neighborhood, each neighbor receives information only about its $(k/2-1)$-hop-neighborhood. Indeed, in the second phase of the convergecast $(k/2-1)$-hop-neighborhoods are considered, so this information from the first phase is sufficient. This requires $O(\lceil \frac{\log|A|}{\log n} \rceil \cdot \Delta^{k/2-1})$ rounds. The other phases require a smaller number of rounds. Thus the overall running time becomes $O(\lceil \frac{\log|A|}{\log n} \rceil \cdot \Delta^{k/2-1})$. We summarize this in the next Corollary.

▶ **Corollary A.3.** *An R round algorithm for G in the CONGEST model that employs a set of commutative idempotent functions $f_1, f_2, ..., f_t : A \to A$ can be transformed into an algorithm for G^k in the CONGEST model, with running time $O(R \cdot \lceil \frac{\log|A|}{\log n} \rceil \cdot \Delta^{k/2-1})$.*

We also obtain an improvement in the memory complexity, as shown in the next corollary.

▶ **Corollary A.4.** *Denote by M the local memory of a vertex needed to complete an algorithm in a straightforward way for G^k, $k \geq 2$, in the CONGEST model. The local memory required* *for an algorithm which uses our aggregation functions, such that* $\log|A| = O(\log n)$*, is bounded by* $O(\frac{M}{\Delta^{k/2}}) \sim O(\sqrt{M})$*.*

Memory considerations as in Corollary A.4 are usually not analyzed in the CONGEST model. However, in real life applications that consider RAM limitations, our technique provides an improvement, which can make algorithm implementation much more efficient.

A.1 Computations with non-idempotent functions

A function that is broadly used by us, and is not idempotent is the counting function. Specifically, given a value that a vertex holds, it needs to compute how many times this value appears in the k-hop-neighborhood of the vertex. It is applied on each vertex in the k-hop-neighborhood, and increments the result by one, each time it encounters the given value. (It does not change the result when it is applied on a different value.) Hence, the number of function applications may affect the result, and thus it is non-idempotent. Nevertheless, we are still interested in using such a function, since it appears in Linial's algorithm for computing number of polynomial intersections, and in arbdefective-coloring for computing the number of vertices of the k-hop-neighborhood with the same color.

There are two options to address this goal. The first option is analysing the solution with the possibility of over-counting. In this case the result is a correct upper bound on the desired solution, and the goal is analyzing how far it is from a solution that counts exactly. The second option is to obtain exact counting, using a more sophisticated construction, which we describe in the sequel.

In the first option we perform an adaption from G to G^k as in the case of idempotent functions, described in Section 4.2. Now we analyze the maximum number of function invocations in an iteration. Note that each invocation corresponds to a unique path of length at most k that starts from a vertex v, goes through a vertex w in the middle of the path, and ends in a vertex in the $k/2$-hop-neighborhood of w. The number of such paths is bounded by $\Delta^k + \Delta^{k-1} + \Delta^{k-2} + ...\Delta = O(\Delta^k)$. Denote this bound by d_p. Now we can use d_p instead of Δ^k in the computations for k-distance-Linial algorithm or arbdefective colorings to distance k. (Note that Δ is known to all vertices, so they can compute $d_p = \Delta^k + \Delta^{k-1} + \Delta^{k-2} + ...\Delta$.) These computations will provide correct results, according to the same pigeonhole principle, as in the proof of the distance-2 variants, but the results now depend on d_p, rather than Δ^k.

We illustrate this with an adaptation of Linial's algorithm to provide distance-k coloring with $O(\Delta^{2k})$ colors, for a constant k. Let $d_p = O(\Delta^k)$, as described above. We compute a coloring with $O(d_p^2) = O(\Delta^{2k})$ colors. This is done by $O(\log^* n)$ stages, each of which perform a binary search for $O(\log \Delta^k) = O(\log \Delta)$ phases. The binary search is done using the aggregation counting function, to count the number of conflicts in distance k of a polynomial P, for each half of its range. Even though this may cause over-counting, the aggregation functions are applied at most d_p times per polynomial per phase. Thus using a field of size q, $d_p < q < 2d_p$, guarantees that there is going to be $t \in \{0, 1, ..., q-1\}$, such that $P(t)$ does not intersect with any polynomial in its k-hop-neighborhood. Such value t is going to be found by the binary search. This results in an $O(d_t^2)$-coloring in $O(\Delta^{k/2-1} \cdot \log d_t \cdot \log^* n)$ time, i.e., $O(\Delta^{2k})$-coloring in $O(\Delta^{k/2-1} \cdot \log \Delta \cdot \log^* n)$ time. We summarize this in the next corollary.

▶ **Corollary A.5.** *For a constant* $k \geq 1$*, we compute* $O(\Delta^{2k})$*-coloring of* G^k *in* $O(\Delta^{k/2-1} \cdot \log \Delta \cdot \log^* n)$ *rounds in the CONGEST model.*

While the above option is a simple solution, it provides an upper bound, which may be several times larger that the exact value. Also, for other functions, applying them more than the required number of times may not work. Thus we propose a second option that allows executing the function exactly the desired number of times. This requires a one-time preprocessing stage with $O(\Delta^{k-1})$ rounds. Still, for algorithms with running time R in G of at least $\Delta^{1/2}$, this does not affect significantly the overall running time of the transformation for G^k. (Recall that the best currently-known deterministic algorithm for $O(\Delta)$-coloring and MIS have running times $\tilde{O}(\Delta^{1/2} + \log^* n)$ and $O(\Delta + \log^* n)$, respectively.)

In the preprocessing stage, each vertex $v \in V$ computes a BFS tree of height $k/2$ rooted at v. (Thus, the radius of the tree is also bounded by $k/2$.) This is the BFS tree of the $k/2$-hop-neighborhood of v. Then each vertex receives the BFS trees of its $k/2$-hop-neighborhood. Hence, information about k-hop-neighborhoods is obtained. The construction of BFS trees proceeds in $k/2$ phases. In each phase $i = 1, 2, ..., k/2$, BFS trees of height i are constructed from BFS trees of height $i-1$, as follows. A vertex receives the BFS trees of height $i-1$ from its neighbours. It constructs locally a graph G_v, consisting of all vertices of all these trees, and all edges of the trees. (Note that such a composition of trees may cause cycles.) Now a local BFS algorithm is executed on G_v, starting from v. Note that in the graph G_v all edges containing v connect it to roots of trees of height $i-1$. Consequently, the BFS of G_v results in a tree of height i. It is broadcasted to the neighbours of v in phase i. Since a BFS tree of height i may have up to $O(\Delta^i)$ vertices and edges, the running time of phase i is $O(\Delta^i)$. The overall running time for computing BFS trees of height $k/2$ is $O(\Delta^{k/2-1})$. This is because in the beginning of phase $k/2$, BFS trees of height $k/2-1$ are received, which requires $O(\Delta^{k/2-1})$ rounds. Then BFS trees of height k are constructed locally, which completes the computation of the BFS tree.

Next, for each vertex $v \in V$ in parallel, the BFS tree that v computed is sent to all vertices in the $k/2$-hop-neighborhood of v. This is done by parallel broadcasting within $k/2$ phases. Phase 1 requires $O(\Delta^{k/2})$ rounds, phase 2 requires $O(\Delta^{k/2+1})$ rounds, etc. Phase $k/2$ requires $O(\Delta^{k-1})$ rounds, which is also the overall time of the preprocessing stage.

Next, we explain how these BFS trees are used to make sure that each function invocation in the convergecast stage of our algorithm is performed exactly once, for each vertex in the k-hop-neighborhood of v. Let $w \in V$ be a vertex that performs the convergast stage, and v, u be vertices in the $k/2$-hop-neighborhood of w. The vertex w has to decide whether to invoke a function of v on u. Since w holds the BFS tree of height $k/2$ of v, the vertex w knows who are all the other vertices w' that also considering at that stage whether to execute a function for v. Moreover, w knows the set $W' = \{w' \mid w'$ is a vertex at distance $k/2$ from v, and at distance at most $k/2$ from $u\}$. This is because w holds all BFS trees of its $k/2$-hop-neighborhood. Now, w can find the vertex with the smallest ID in W'. If this is w itself it applies the function, and does not apply it otherwise. This way, out of all vertices w' that could have applied it, exactly one vertex does so.

It remains to address vertices u at distance less than $k/2$ from v, for which there are no vertex w at distance $k/2$ from v, such that u is in the $k/2$-hop-neighborhood of w. This is addressed directly by v, using the BFS tree of v. Since the BFS tree of w is also known to v, the vertex v can decide to apply the function on such u that are not in the BFS tree of w.

Now, our scheme that described in Section 4.2 for computations on G^k with idempotent functions can be extended to non-idempotent functions. The difference in the generalized scheme is that it starts with the preprocessing stage of computing BFS-trees of height $k/2$. In addition, during the main stages of the scheme, whenever a function has to be applied, a check is performed to ensure that it is applied exactly once for a pair v, u, as explained above. This gives rise to the following result.

▶ **Theorem A.6.** *Suppose we are given a set A of all possible inputs and outputs of aggregation functions, and an R-round algorithm for G in the CONGEST setting, where each round* *consists of a series of invocations of commutative aggregation functions* $f_1, f_2, ...f_t : A \to A$*. Then we can obtain an algorithm for the same problem on* G^k *whose running time in the CONGEST model is* $O(R \cdot \lceil \frac{\log |A|}{\log n} \rceil \cdot \Delta^{k/2-1} + \Delta^{k-1})$.

A.2 Applications of the speedup technique for G^k

In this section we provide several application of Theorem A.6. Consider the $O(\Delta)$-coloring algorithm for G on which the results of Section 3 are built. It consists of three parts:

(1) Computing defective coloring. This part employs an aggregation function to compute number of intersections in Linial's algorithm.

(2) Computing arbdefective coloring. This part employs an aggregation function to compute the number of neighbors with the same color.

(3) Iterating over color classes and computing a proper coloring iteratively. This part employs an aggregation function that again computes the number of polynomial intersections, but this time according to the algorithm in Section 3.2.3.

In all these functions we can set $A = \{1, 2, ..., n\}$, since the results are always bounded by the number of vertices in the graph. Since the algorithm for G composed of these steps requires $\tilde{O}(\Delta^{1/2} + \log^* n)$ rounds [9], it gives rise to an $\tilde{O}(k \cdot \Delta^{k-1} + k \cdot \Delta^{\frac{k}{2}} \log^* n)$-round algorithm for G^k, by Theorem A.6. (The factor of k appears in the running time because the $O(\log \Delta)$ running time for binary searches in G translates into $O(\log \Delta^k) = k \log \Delta$ phases in G^k.) This is summarized in the next corollary.

▶ **Corollary A.7.** *For* $k > 1$*, an* $O(\Delta^k)$*-coloring of* G^k *can be computed in* $\tilde{O}(k \cdot \Delta^{k-1} + k \cdot \Delta^{\frac{k}{2}} \log^* n)$ *rounds in the CONGEST model.*

Next, we obtain an MIS algorithm for G^k. It starts with computing $O(\Delta^k)$-coloring of G^k. Then, for each color class $i = 1, 2, ...,$ an iteration is performed, consisting of k rounds. Specifically, all vertices of color i perform broadcast to distance k. That is, each such vertex initializes a broadcast message with a counter that is initialized to 0. Each time such a message is received, the counter is incremented by 1. If a vertex receives several messages in parallel, it handles only one of them, and the others are discarded. If the counter of the message is smaller than k it is sent to the immediate neighbors. Otherwise, it is discarded. Therefore, an iteration completes within k rounds. Then vertices of color i that initiated the broadcast decide to join the MIS, and vertices that received the messages decide not to join. Vertices that made decisions become inactive. Then the next iteration starts on the remaining active vertices, etc. After $O(\Delta^k)$ such iterations, the algorithm terminates.

▶ **Theorem A.8.** *MIS of* G^k *can be computed within* $\tilde{O}(k \cdot \Delta^k + k \cdot \Delta^{k-1} \cdot \log^* n)$ *rounds in CONGEST model.*

Proofs and details of section A.2 appear in the full version of this paper [10].

A Fully Concurrent Adaptive Snapshot Object for RMWable Shared-Memory

Benyamin Bashari
University of Calgary, Canada

David Yu Cheng Chan
University of Calgary, Canada

Philipp Woelfel
University of Calgary, Canada

Abstract

An adaptive RMWable snapshot object maintains an array $A[0..m-1]$ of m readable shared memory objects that support an arbitrary set of read-modify-write (RMW) operations, in addition to `Read()`. Each array entry $A[i]$ can be accessed by any process using an operation `Invoke(`i, op`)`, which simply applies a supported RMW operation op to $A[i]$ and returns the response of op. In addition, processes can record the state of the array by calling `Click()`. While `Click()` does not return anything, a process p can call `Observe(`i`)` to determine the value of $A[i]$ at the point of p's latest `Click()`.

Recently, Jayanti, Jayanti, and Jayanti [10] presented an RMWable adaptive snapshot object, where all operations have constant step complexity. Their algorithm is *single-scanner*, meaning that `Click()` operations cannot be executed concurrently. We present the first *fully concurrent* RMWable adaptive snapshot object, where all operations can be executed concurrently, assuming the the system provides atomic Fetch-And-Increment and Compare-And-Swap operations. `Click()` and `Invoke()` operations have constant step complexity, and `Observe()` has step complexity $O(\log n)$. The total number of base objects needed is $O(mn \log n)$.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases Shared memory, snapshot, camera object, RMW, distributed computing

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.7

Funding We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC),under Discovery Grant RGPIN/2019-04852, and the Canada Research Chairs program.

1 Introduction

Linearizable snapshot objects are a fundamental building block for shared memory algorithms. A snapshot object maintains an array of m registers, $A[0 \ldots m-1]$. The standard definition allows a process to write to an array entry, and to perform a `Scan()`, which returns the vector $(A[0], \ldots, A[m-1])$. Most research considers single-writer snapshots, where m is equal to the number of processes, n, and process i can only write to $A[i]$.

Implementing deterministic linearizable single-writer snapshot objects from atomic registers (which support read and write operations) has been studied intensively (e.g., [6, 1, 2, 8]). Inoue and Chen [8] devised a linearizable snapshot, where each operation has at most linear step complexity, which is optimal at least for `Scan()` operations [12]. In order to circumvent this lower bound, researchers limited the number of operations [3] or employed randomization [4, 13].

Many snapshot algorithms assume that the size of a memory word is large enough to store the entire state of array A. This is an unrealistic assumption, unless large registers are simulated by smaller ones, which is inherently inefficient. Employing stronger primitives, such as compare-and-swap (CAS) and fetch-and-increment (FAI) objects, one can obtain

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 7; pp. 7:1–7:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

snapshot objects, where it is sufficient for a memory location to store a single array entry [15]. However, then the complexity of a `Scan()` is inherently lower bounded by the size of the array.

To deal with this inherent inefficiency, some researchers studied snapshot types that allowed certain operations to return limited information about array A more efficiently. For example, Jayanti [9] proposed the *f-array* object, where a `Read()` operation returns the value of a function f applied to all components of array A. This function can be computed in a constant number of steps, but updating array A is more expensive: In Jayanti's original algorithm (which allows read-modify-write (RMW) operation to be applied to individual components of A) updating a single component of A has step complexity $\Theta(m)$, where m is the size of the array. Obryk [14] provided a version of this object, where components can only be updated with write operations, but in $O(\log^3 m)$ steps.

Attiya, Guerraoui, and Ruppert [5] followed a different approach: Their *partial snapshot object* allows processes to obtain a view of only some of the entries of A. The step complexity of such a *partial scan* is quadratic in the number of array entries the view contains, and the amortized step complexity of updates is bounded by the maximum interval contention, as well as the maximum number of components accessed by partial scan operations. Bashari and Woelfel [7] devised an *adaptive* single-writer snapshot object, where a snapshot is taken by a `Click()`[1] operation that does not return anything. Instead, a process can later determine the value of any array entry $A[i]$ at the point of its latest preceding `Click()`, by performing an `Observe(`i`)` operation. Contrary to the partial snapshots of Attiya, Guerraoui, and Ruppert [5], this semantics allows observed array entries to be chosen adaptively, based on previously observed values. The algorithm uses polynomially many single-word registers and CAS objects, as well as an unbounded FAI object. `Click()` has constant step-complexity, whereas updating or observing an array entry takes $O(\log n)$ steps.

Another shortcoming of many snapshot algorithms is that the entries of array A can only be updated with write operations. But modern shared memory systems critically support many types of read-modify-write (RMW) operations, which are much more powerful than reads and writes, and most non-trivial data structures rely on such RMW operations. Thus, conventional snapshot algorithms (where write is the only allowed update operation) cannot be used to obtain snapshots of most data structures. Jayanti's f-array object [9] addresses this issue, by allowing the components of array A to be of arbitrary types. But, as mentioned earlier, updates have step complexity of $\Omega(m)$.

Wei, Ben-David, Blelloch, Fatourou, Ruppert and Sun [16] also presented a snapshot object, where the array entries can be modified with `CAS()` operations. The algorithm supports a snapshot operation that returns a *handle*. The value of individual array entries at the point of when the handle was obtained, can then be inspected adaptively. The algorithm uses CAS objects, and the step complexity of observing the value of a single array entry grows linearly with the number of updates that may have occurred on that location, since the corresponding snapshot was taken. The authors also showed that their interface can be used to easily add snapshot operations to concurrent data structures (that are implemented from CAS objects), and presented experimental results, indicating a low overhead of this approach.

Very recently, Jayanti, Jayanti, and Jayanti [10] presented an *RMWable* adaptive snapshot object. Their algorithm generalizes the semantics of Bashari and Woelfel's adaptive snapshot object, by allowing array entries to be updated with any RMW operations [10] that are

[1] This operation was also called `Scan()` in [7]. Jayanti, Jayanti, and Jayanti [10] used the term `Click()`, which more clearly indicates that the semantics is different from a standard `Scan()`.

supported by the system. Their algorithm has optimal constant step complexity for `Click()`, but multiple `Click()` operations cannot be executed concurrently. We present a (completely different and independently devised) algorithm for the same sequential specification. Our algorithm achieves full concurrency (i.e., it allows concurrent `Click()` operations) for the price of `Observe()` operations having a step complexity $O(\log n)$ instead of constant.

Consider a set $\mathcal{O}$ of wait-free linarizable objects available to the system, such that each object supports a read operation (among others). Our adaptive RMWable snapshot object maintains an array A of m objects from $\mathcal{O}$, where m is an arbitrary positive integer. (The assumption that all array components are of the same type is made for ease of description only; in fact, each array entry can be of a different readable type.)

Each process p can execute `Invoke(`i`, `op`)` to apply any operation op (supported by the object represented by $A[i]$) to $A[i]$ and obtain the response of that operation. A process can take a snapshot of the array using a `Click()` operation, which returns nothing. Finally, p can at any point call `Observe(`i`)`, which returns the value of $A[i]$ at the point of p's latest `Click()` operation.

We assume that the system provides atomic FAI and CAS operations. In a system with n processes, `Click()` and `Invoke()` operations have constant step complexity, and `Observe()` has step complexity $O(\log n)$. The total number of base objects needed is $O(mn \log n)$.

The FAI object needs to perform approximately one increment per implemented operation, and the resulting values need to be stored in other objects. Thus, strictly speaking, our algorithm can only perform a bounded number of operations. However, in practice this bound will never be reached on 64-bit architectures.

In the following section we describe the system model and specify the object we are implementing. Then, in Section 3 we present the algorithm and its properties. Finally, in Section 4, we will proof correctness. The analysis of time and space complexity is omitted due to space restrictions.

2 Preliminaries

We consider the standard asynchronous shared memory model with n processes with IDs $0, \ldots, n-1$, which communicate using atomic (or linearizable) shared memory operations on *base objects*.

A register supports the standard `Read()` and `Write()` operations. An LL/SC object provides operations, `LL()` and `SC(`v`)`, where `LL()` returns the object's value, and `SC(`v`)` called by process p updates the value to v, if p has previously called `LL()` and no successful `SC()` operation has occurred since then. An `SC()` operation returns a Boolean value indicating if it successfully stored its parameter. A FAI object stores an integer, initially 1, and provides an operation `FAI()`, which increments the object's value by 1 and returns the value before the increment.

While `FAI()` is available on standard hardware, LL/SC is not. However, there are efficient implementations of LL/SC from registers and CAS objects, which are usually available. For example, by using unbounded sequence numbers, one can implement an LL/SC object from a single CAS object with constant step complexity. An algorithm by Jayanti and Petrovic [11] avoids unbounded sequence numbers, but needs $O(n)$ CAS objects to implement an LL/SC object with constant step complexity.

The Adaptive RMWable Snapshot Object

Let $\mathcal{O}$ denote a set of wait-free linearizable objects that are available in the system. Each object in that set must be readable, i.e., support an operation that returns the state of the object without changing it. For the ease of description, we assume that each operation on such an object takes at most a constant number of steps.

The adaptive RMWable snapshot object maintains an array of m components, each corresponding to an object in $\mathcal{O}$ in its initial state. For convenience, we assume w.l.o.g. that the initial state of each component is 0.

An adaptive RMWable snapshot object allows each process p to perform the following operations:

- `Invoke(`i, op`)` performs operation op (which must be one of the operations supported by $\mathcal{O}$) on the i-th component , and returns the corresponding response.
- `Click()` simply returns **done** (this is convenient for our proofs, but equivalently, one may assume that it returns nothing).
- `Observe(`i`)` returns the value of component i at the time of p's last `Scan()`; or the initial state 0 of component i if no such `Scan()` exists.

3 The Algorithm

Let n and m be positive integers, κ be a sufficiently large constant, $\Delta' = O(\log n)$, and $\Delta = \kappa n \log n/\Delta'$. In this section, we present an implementation of the adaptive RMWable snapshot object for n processes and m components such that:

- The space complexity of the implementation is $O(m\Delta)$.
- The time complexity of `Click()` operations is $O(1)$.
- The time complexity of `Invoke()` operations is $O(\Delta')$.
- The time complexity of `Observe()` operations is $O(\log \Delta)$.

Thus if we select $\Delta' = 1$, we have $\Delta = O(n \log n)$ and thus obtain:

- The space complexity of the implementation is $O(mn \log n)$.
- The time complexity of `Click()` operations is $O(1)$.
- The time complexity of `Invoke()` operations is $O(1)$.
- The time complexity of `Observe()` operations is $O(\log n)$.

3.1 Bashari and Woelfel's Single-Writer Snapshot

The fundamental idea of our algorithm is based on Bashari and Woelfel's adaptive partial snapshot algorithm [7]. Their algorithm implements an adaptive snapshot object for n processes and $m = n$ components that each correspond to a single writer register. Hence, instead of `Invoke(`$i, -$`)`, it supports `Write(`i, val`)`, which only process i can execute in order to write some value val to the i-th component. Their algorithm employs a FAI object clk, and m single-writer multi-reader red-black trees. The i-th red-black tree can only be updated by process i, who uses it to record the past states of component i. On a high level, the algorithm works as follows:

- Each `Click()` operation takes a timestamp from the FAI object clk.
- Each `Write()`i, val operation takes a timestamp from the FAI object clk. Then it simply stores val along with its timestamp into the i-th red-black tree.
- Each `Observe(`i`)` operation by a process p searches the i-th red-black tree for the state with the largest timestamp that is smaller than the timestamp of the latest `Click()` operation by process p.
- The red-black trees are periodically pruned of recorded states that are no longer necessary, and thus inserts and searches take only $O(\log n)$ steps.

The i-th red-black tree serves as a predecessor data structure that can be queried by all processes but only updated by process i. As our algorithm allows updates on component i to be performed by any process, we need to replace each red-black tree with a multi-writer predecessor data structure. Moreover, adding the correct elements to the predecessor data structure is substantially more challenging, because multiple processes may perform `Invoke`$(i, -)$[2] concurrently.

3.2 Outline of our Algorithm

Algorithm 1 depicts our adaptive RMWable snapshot implementation. Similar to Bashari and Woelfel we use an FAI object *clk* to record timestamps. Consider some $i \in \{0, \ldots, m-1\}$. We use an object $O[i]$ with the same sequential specification as the i-th component object. To perform `Invoke`$(i, \mathtt{op}_i)$, a process p performs operation op_i on $O[i]$ and records the return value, which it will later use as its response. Before p's `Invoke`$(i, \mathtt{op}_i)$ can linearize, the resulting state of component i needs to be "recorded" in a predecessor data structure, together with a timestamp obtained from *clk*.

The predecessor data structure for the i-th component is implemented using a circularly sorted array $R[0 \ldots \Delta - 1][i]$. For now assume that at most Δ `Invoke`$(i, -)$ operations can be performed; this will ensure that $R[0 \ldots \Delta - 1][i]$ remains completely sorted.

First consider the simplified single-updater case, in which only one process p is allowed to call `Invoke`(i, op_i). In its j-th `Invoke`(i, op_i) operation, after performing op_i on $O[i]$, p can obtain a new timestamp k using a `FAI()` operation on *clk*, and then write k and the new value of $O[i]$ into $R[j][i]$. This way, $R[0 \ldots \Delta - 1][i]$ remains sorted (by timestamp values). A process q that performs a `Click()` also obtains a timestamp k' from *clk*. To observe component i, q can then simply return the value of $O[i]$ that was recorded in the array entry $R[j][i]$, $j \in \{0, \ldots, \Delta - 1\}$, with the largest timestamp $k \leq k'$. That array entry can be found in $O(\log \Delta)$ steps using a binary search.

In order to support multiple concurrent `Invoke`$(i, -)$ operations, processes with pending such operations will agree on *some state* of $O[i]$, and add that agreed upon value to an appropriate array entry of $R[0 \ldots \Delta - 1][i]$, together with an appropriate timestamp k. This is done in a `HelpUpdate()` method, as follows: We use an LL/SC object $lastUpdate[i]$, which stores a triple (j, k, val) where j is a sequence number, k is either a timestamp or $\perp$, and val is either a state of $O[i]$ or $\perp$. Initially, $lastUpdate[i] = (0, \perp, \perp)$. In `HelpUpdate()`, a process q repeats the following several times: If $lastUpdate[i] = (j, \perp, \perp)$ then it reads the current value val from $O[i]$ and tries to change $lastUpdate[i]$ to $(j+1, \perp, val)$ using an `SC()` operation. If $lastUpdate[i] = (j, \perp, val)$ for $val \neq \perp$, then q obtains a timestamp k from *clk* and tries to change $lastUpdate[i]$ to (j, k, val). Once $lastUpdate[i] = (j, k, val)$ for $k, val \neq \perp$, the pair (k, val) is the agreed upon pair that will be added to the predecessor object. Since (k, val) is the j-th agreement pair, it can simply be written to $R[j][i]$. Nothing changes for `Click()` and `Observe()` operations.

To see that this is linearizable, consider the following linearization points: A `Click()` linearizes when the calling process obtains a timestamp from *clk*, and an `Observe()` can linearize at any point during its execution interval. Now consider an `Invoke`(i, op_i) operation during which process p performs op_i on $O[i]$ at some point t. Let t' be the first point after t, at which the value of $O[i]$ is copied to $lastUpdate[i]$. Then p's `Invoke`(i, op_i) linearizes at

[2] Throughout this text we use a dash ("$-$") as the argument of a method call, to indicate that the statement applies to all arguments.

Algorithm 1 Adaptive RMWable Snapshot Implementation.

```
Shared:
    FAI $clk$, initially 1
    LL/SC $lastScan[0 \dots n-1][0 \dots m]$, each initially $(0,0,0,0,0)$
    LL/SC/Read $lastUpdate[0 \dots m-1]$, each initially $(0,0,\perp)$
    Object $O[0 \dots m-1]$, each initially fresh.
    LL/SC $R[0 \dots \Delta-1][0 \dots m-1]$, initially $(0,0,0)$
Code for each process $p$:
Function Click()
    $(k,-,-,-,v) \leftarrow lastScan[p][m]$.LL()
    if $v = 0$ then $lastScan[p][m]$.SC($k, 0, 0, 0, 1$)
    HelpScan($p$)
    return done
Function HelpScan($q$)
    $(-,-,-,-,v) \leftarrow lastScan[q][m]$.LL()
    if $v = 1$ then
        $k \leftarrow clk$.FAI()
        $lastScan[q][m]$.SC($k, 0, 0, 0, 0$)
Function Observe($i$)
    HelpUpdate($i$)
    repeat $v \leftarrow$ HelpObserve($p, i$) until $v \neq \perp$
    return $v$
Function HelpObserve($q, i$)
    $(k_i, maxKey, j_{left}, j_{right}, v) \leftarrow lastScan[q][i]$.LL()
    $(k_m,-,-,-,-) \leftarrow lastScan[q][m]$.LL()
    if $k_m > k_i$ then
        $(j_u, k_u, v_u) \leftarrow lastUpdate[i]$.Read()
        if $(k_u, v_u) \neq (0,\perp)$ then $j_{right} \leftarrow j_u + \Delta - 1 \bmod \Delta$
        else $j_{right} \leftarrow j_u \bmod \Delta$
        $j_{left} \leftarrow j_{right} + 1 \bmod \Delta$
        $(-, maxKey, -) \leftarrow R[j_{right}][i]$.LL()
        $(k_i, v) \leftarrow (k_m, \perp)$
        if $maxKey < k_i$ then return $\perp$
    else
        if $v \neq \perp$ then return $v$
        if $j_{left} > j_{right}$ then $j \leftarrow \lceil (j_{left} + j_{right} + \Delta)/2 \rceil \bmod \Delta$
        else $j \leftarrow \lceil (j_{left} + j_{right})/2 \rceil$
        $(-, k_r, -) \leftarrow R[j][i]$.LL()
        if $k_r \geq k_i$ and $k_r \leq maxKey$ then $j_{right} \leftarrow j + \Delta - 1 \bmod \Delta$
        else $j_{left} \leftarrow j$
        if $j_{left} = j_{right}$ then $(-,-,v) \leftarrow R[j_{left}][i]$.LL()
    $lastScan[q][i]$.SC($k_i, maxKey, j_{left}, j_{right}, v$)
    return $\perp$
Function Invoke($i, op_i$)
    $v_{res} \leftarrow O[i].op_i()$
    HelpUpdate($i$)
    return $v_{res}$
Function HelpUpdate($i$)
    for $a \in \{0, \dots, 5\}$ do
        $(j_u, k_u, v_u) \leftarrow lastUpdate[i]$.LL()
        if $v_u = \perp$ then
            $v \leftarrow O[i]$.Read()
            $lastUpdate[i]$.SC($j_u + 1, \perp, v$)
        else
            if $k_u = \perp$ then
                $k \leftarrow clk$.FAI()
                $lastUpdate[i]$.SC($j_u, k, v_u$)
            else
                $(j,-,-) \leftarrow R[j_u \bmod \Delta][i]$.LL()
                if $j < j_u$ then $R[j_u \bmod \Delta][i]$.SC($j_u, k_u, v_u$)
                for $a' \in \{0 \dots \Delta'\}$ do
                    HelpScan($j_u \bmod n$)
                    HelpObserve($j_u \bmod n, i$)
                $lastUpdate[i]$.SC($j_u, 0, \perp$)
```

the first point when some process obtains a sequence number k from clk, such that the pair (k, val) gets stored in $lastUpdate[i]$. In other words, if (j, k_j, val_j) is the j-th triple stored in $lastUpdate[i]$ satisfying $val_j, k_j \neq \perp$, then all `Invoke`$(i, -)$ operations whose operation on $O[i]$ is reflected in val_j but not val_{j-1} linearize at the point timestamp k_j is obtained. The essential steps of `HelpUpdate()` are repeated sufficiently many times to ensure that this happens before any of the linearized `Invoke`$(i, -)$ methods respond.

In the above approach, `HelpUpdate()` allows processes to repeatedly agree on a value $O[i]$ and an associated timestamp. If (k_j, val_j) is the j-th agreed timestamp-value pair, then the triple (j, k_j, val_j) will be written to $R[j][i]$. As $R[0 \ldots \Delta - 1][i]$ has size Δ, this only works if the number of `Invoke`$(i, -)$ operations is bounded by Δ. To support an unbounded number of `Invoke`$(i, -)$ operations, the triple (j, k_j, val_j) will be written to $R[j \bmod \Delta][i]$, instead. While the array remains circularly sorted, and binary search is still possible, we now face the problem that old values in $R[0 \ldots \Delta][i]$ will eventually get overwritten.

We deal with that as follows: Following a `Click()` call by process p, for each $i \in \{0, \ldots, m-1\}$, the relevant value stored in $R[0 \ldots \Delta][i]$ (i.e., the one which p would have to return in a subsequent `Observe`(i) operation), will be copied to another LL/SC object, $lastScan[p][i]$. When some process q performs `HelpUpdate`(i), it contributes $O(\Delta')$ of work to that, guaranteeing that all relevant array entries of $R[0 \ldots \Delta][i]$ are copied to $lastScan[p][i]$, before they get overwritten. It does so by calling `HelpObserve`(p, i). In that method call, it contributes a constant number of steps to a binary search on $R[0 \ldots \Delta][i]$ for the relevant array entry. To facilitate multiple processes participating in this binary search, $lastScan[p][i]$ stores a 5-tuple $(k_i, maxKey, j_{left}, j_{right}, v)$, where k_i is the timestamp that p obtained during its `Click()`, $maxKey$ is essentially the largest key found in $lastUpdate[i]$, when the first process started the binary search, j_{left} and j_{right} are the current left and right borders found during the binary search, and v will eventually be set to the correct value (representing the state of $O[i]$) found in the binary search. Each process q contributes to the binary search by loading the value of $lastScan[p][i]$, computing the next value that needs to be written to $lastScan[p][i]$, and then attempting to write that value using an `SC()` operation. If some other process has already performed that next step of the binary search, then q's `SC()` will simply fail. The exact details of the binary search are described in Section 3.3.

We still need to deal with one other problem: Suppose process p obtains a timestamp k from clk in its `Click()` method, and immediately after that falls asleep, before it can write k anywhere. Then the relevant value of $R[0 \ldots \Delta][i]$ may get overwritten before any other process even learns about k. I.e., no process can help copying relevant values from $R[0 \ldots \Delta][i]$ to $lastScan[p][i]$, before it's too late. To deal with that, at the beginning of its `Click()`, process p announces that it has started a `Click()` operation by setting a bit in the last component of $lastScan[p][m]$. (Note the index m, which means the array entry is not used for values copied from R.) That bit indicates that other processes should help p with its `Click()` operation, specifically with obtaining and publishing a timestamp. They do so by calling a method `HelpScan`(p) before each `HelpObserve`(p) call during `HelpUpdate()` (the helped process, p is chosen in a round-robin fashion, based on the sequence number found in $lastUpdate[i]$). In such a `HelpScan`(p) call, process q checks if p wants help (as indicated by the last component of $lastScan[p][m]$), and if yes, q obtains a timestamp k from clk. Then, using an `SC()` operation, it tries to store that timestamp into the first component of $lastScan[p][m]$ while also resetting the last component to 0. The timestamp associated with p's `Click()` operation is then the first timestamp that gets written to $lastScan[p][m]$, and the `Click()` linearizes when that timestamp is obtained from clk.

3.3 Low Level Description

Our algorithm uses the following shared objects:

- *clk*: A FAI object that stores timestamps.
- $\mathit{lastScan}[0 \ldots n-1][0 \ldots m]$: An array of LL/SC objects that record the timestamps of the last `Click()` operation by each process and the states of each component object at the time when the timestamp was received from *clk*.

  For every process $p \in \{0, 1, \ldots, n-1\}$, $\mathit{lastScan}[p][m]$ stores a tuple $(k, 0, 0, 0, v)$, where k is the timestamp of the last `Click()` operation by process p, and $v = 1$ if p needs another timestamp for pending `Click()` operation; otherwise $v = 0$.

  For every process $p \in \{0, 1, \ldots, n-1\}$ and every integer $i \in \{0, 1, \ldots, m-1\}$, $\mathit{lastScan}[p][i]$ stores a tuple $(k, \mathit{maxKey}, j_{\mathit{left}}, j_{\mathit{right}}, v)$, where k is the last detected timestamp of the last `Click()` operation by process p, v is either the state of the i-th component object at the time when k was received from *clk* or $\perp$ if that is yet to be deduced, and maxKey, j_{left}, and j_{right} are integers that are used to help deduce that state.
- $O[0 \ldots m-1]$: For every integer $i \in \{0, 1, \ldots, m-1\}$, $O[i]$ is a wait-free linearizable readable base object with the same sequential specification as the i-th component object, and is used to determine the state of this i-th component object at various timestamps. Note that at any time, the state of the i-th component object is not necessarily the same as the state of $O[i]$.
- $\mathit{lastUpdate}[0 \ldots m-1]$: For every integer $i \in \{0, 1, \ldots, m-1\}$, $\mathit{lastUpdate}[i]$ is an LL/SC object that is intuitively used to repeatedly pair a timestamp k received from *clk* with a state v read from the base object $O[i]$, and thus intuitively set the state of the i-th component object to v at the time when timestamp k was received from *clk*.
- $R[0 \ldots \Delta-1][0 \ldots m-1]$: For every integer $i \in \{0, 1, \ldots, m-1\}$, $R[0 \ldots \Delta-1][i]$ is a circularly sorted array of LL/SC objects that records the previous states for the i-th component object (replacing the red-black trees of [7]).

  For every integer $i \in \{0, 1, \ldots, m-1\}$ and $j \in \{0, 1, \ldots, \Delta-1\}$, $R[j][i]$ stores a tuple (j_r, k_r, v_r), where roughly speaking, k_r is a timestamp, v_r was the state of the i-th component at the time when k_r was received from *clk*, and (j_r, k_r, v_r) was the value in $\mathit{lastUpdate}[i]$ at the time when $R[j][i]$ was last modified.

To achieve the desired time and space complexities, our algorithm heavily relies on various helping mechanisms, which we have divided into the auxiliary functions `HelpScan(`q`)`, `HelpObserve(`q, i`)`, and `HelpUpdate(`i`)` that intuitively help to complete `Click()`, `Observe()`, and `Invoke()` operations respectively.

In the following we describe which steps a process p performs during each of the indicated operations.

3.3.1 `HelpScan()`

During each `HelpScan(`q`)` operation, a process p performs the following steps:

1. It performs an `LL()` operation on $\mathit{lastScan}[q][m]$ to check whether process q needs a timestamp for a pending `Click()` operation (line 7).
2. If so, it takes a timestamp k from the FAI object *clk* (line 9), and attempts to give this timestamp to q's pending `Click()` operation by performing an `SC(`k, *0*, *0*, *0*, *0*`)` operation on $\mathit{lastScan}[q][m]$ (line 10).

3.3.2 HelpObserve()

During each HelpObserve(q, i) operation, a process p performs the following steps:

1. It performs an LL() operation on $\mathit{lastScan}[q][i]$ to get a tuple $(k_i, \mathit{maxKey}, j_{\mathit{left}}, j_{\mathit{right}}, v)$, which indicates the prior progress (if any) that has been made in helping a potential Observe(i) operation by process q after its last Click() operation (line 16).
2. It performs an LL() operation on $\mathit{lastScan}[q][m]$ to read the timestamp k_m of the last Click() operation by process q (line 17).
3. If the timestamp in $\mathit{lastScan}[q][i]$ is older than the timestamp k_m in $\mathit{lastScan}[q][m]$, then that indicates that no progress has been made in helping a potential Observe(i) operation by process q after its last Click() operation (line 18).

   In this case, the tuple $(k_i, \mathit{maxKey}, j_{\mathit{left}}, j_{\mathit{right}}, v)$ that was received from $\mathit{lastScan}[q][i]$ is outdated and p has to compute replacement values for them. So p performs the following steps:

   a. It reads $\mathit{lastUpdate}[i]$ (line 19) to determine the integer j_{right} that corresponds to the (first or second) most recent entry of $R[0 \ldots \Delta - 1][i]$ to be modified (lines 20 to 21), and the integer j_{left} that is for the next entry after j_{right}.

   b. It then reads the timestamp maxKey from the entry corresponding to j_{right} (line 23).

   c. It then sets k_i to k_m and v to $\bot$ (line 24).

   d. If the timestamp maxKey is older than the timestamp $k_i = k_m$ that was received from $\mathit{lastScan}[q][m]$, then it is not safe to help any potential Observe(i) operation by q yet. Intuitively, this is because there could still be pending Invoke($i, -$) operations that could potentially be linearized before the last Click() operation by process q. So in this case, p simply returns $\bot$ on line 25, indicating that future help may still be needed.

   e. Otherwise, p attempts to set $\mathit{lastScan}[q][i]$ to $(k_m, \mathit{maxKey}, j_{\mathit{left}}, j_{\mathit{right}}, \bot)$ (line 34), and returns $\bot$ on line 35, indicating that future help may still be needed.

   Otherwise, p performs the following steps:

   a. It checks whether v is a non-$\bot$ value. If so, then this non-$\bot$ value v is already the appropriate value for any potential Observe(i) operation by process q to return, and so there is no more need to help. Thus p simply returns this non-$\bot$ value v on line 27.

   b. It performs a single iteration of a binary search on the circularly sorted array $R[0 \ldots \Delta - 1][i]$, checking the entry that is intuitively the mid-point of j_{left} and j_{right} to compare its timestamp to k_i, and then appropriately setting either j_{left} or j_{right} to the mid-point (lines 28 to 32).

   c. If $j_{\mathit{left}} = j_{\mathit{right}}$, then that indicates that the binary search has completed, and intuitively $R[j_{\mathit{left}}][i]$ should contain $(-, k_r, v_r)$ such that the timestamp k_r is just before the timestamp k_i, and v_r is the state of the i-th component object at the time that the timestamp k_r was received from clk. So in this case, p simply reads $(-, -, v)$ from $R[j_{\mathit{left}}][i]$ (line 33).

   d. Finally, p attempts to set $\mathit{lastScan}[q][i]$ to $(k_i, \mathit{maxKey}, j_{\mathit{left}}, j_{\mathit{right}}, v)$ (line 34), and returns $\bot$ on line 35, indicating that future help may still be needed.

3.3.3 HelpUpdate()

During each HelpUpdate(i) operation, a process p performs the following steps:

1. It performs an LL() operation on $\mathit{lastUpdate}[i]$ to receive a tuple (j_u, k_u, v_u) on line 42. If $v_u = \bot$, then $\mathit{lastUpdate}[i]$ currently contains neither a timestamp from clk nor a state from $O[i]$ (line 43). So p performs the following steps:

   a. It reads a state v from $O[i]$ (line 44).
   b. It performs an $\texttt{SC(}j_u + \mathit{1}, \perp, v\texttt{)}$ operation to store this state v into $\mathit{lastUpdate}[i]$ (line 45).

   Otherwise, if $k_u = \perp$, then $\mathit{lastUpdate}[i]$ currently does not contain a timestamp from clk (line 47). So p performs the following steps:
   a. It takes a timestamp k from clk via a FAI operation (line 48).
   b. It performs an $\texttt{SC(}j_u, k, v_u\texttt{)}$ operation to store this timestamp k into $\mathit{lastUpdate}[i]$ (line 49).

   Otherwise, $\mathit{lastUpdate}[i]$ currently contains both a timestamp k_u from clk and a state v_u from $O[i]$, indicating that the state of the i-th component object was v_u at the time when the timestamp k_u was received from clk. So p performs the following steps:
   a. It performs an `LL()` operation on $R[j_u \bmod \Delta][i]$ (line 51), which intuitively should be the least recent entry of $R[0 \ldots \Delta - 1][i]$ to be modified, which makes it the safest to overwrite.
   b. If this entry has not yet been modified by a concurrent $\texttt{HelpUpdate(}i\texttt{)}$ operation by any other process, then process p performs an $\texttt{SC(}j_u, k_u, v_u\texttt{)}$ operation on $R[j_u \bmod \Delta][i]$ (line 52) to now record that the state of the i-th component object was v_u at the time when the timestamp k_u was received from clk.
   c. It performs $\Delta' + 1$ alternating $\texttt{HelpScan(}j_u \bmod n\texttt{)}$ and $\texttt{HelpObserve(}j_u \bmod n, i\texttt{)}$ operations (lines 53 to 55). Intuitively, this ensures that enough help is given to `Click()` and `Observe()` operations such that the next least recent entries of $R[0 \ldots \Delta - 1][i]$ are no longer needed and can be safely overwritten.
   d. It performs an $\texttt{SC(}j_u, \mathit{0}, \perp\texttt{)}$ operation on $\mathit{lastUpdate}[i]$ to indicate that it is now ready for a new timestamp and state pair (line 56).
2. It repeats from the start another 5 times, which intuitively ensures that enough help is given to `Invoke()` operations such that the resulting state of the i-th component object and a corresponding timestamp is now recorded.

3.3.4 Click()

Each process p performs the following steps to perform a `Click()` operation:
1. It changes the last field of $\mathit{lastScan}[p][m]$ to 1 (line 2), to indicate to all other processes that process p needs a timestamp for this pending `Click()` operation.
2. It calls $\texttt{HelpScan(}p\texttt{)}$ (line 4) to help itself complete this `Click()` operation, then returns **done** (line 5).

3.3.5 Observe()

Each process p performs the following steps to perform an $\texttt{Observe(}i\texttt{)}$ operation:
1. It calls $\texttt{HelpUpdate(}i\texttt{)}$ (line 12) to help complete any pending $\texttt{Invoke(}i, -\texttt{)}$ operations that could interfere with this $\texttt{Observe(}i\texttt{)}$ operation.
2. It repeatedly calls $\texttt{HelpObserve(}p, i\texttt{)}$ to help this $\texttt{Observe(}i\texttt{)}$ operation until it receives a non-$\perp$ value v (line 12), which it then returns (line 14).

3.3.6 Invoke()

Each process p performs the following steps to perform an $\texttt{Invoke(}i, \mathit{op}_i\texttt{)}$ operation:
1. It performs the operation op_i on $O[i]$ (line 37), changing the state of $O[i]$ and receiving an appropriate response value v_{res} for this $\texttt{Invoke(}i, \mathit{op}_i\texttt{)}$ operation.
2. It calls $\texttt{HelpUpdate(}i\texttt{)}$ (line 38) to help to record down a state of the i-th component and a timestamp into $R[0 \ldots \Delta - 1][i]$, then returns v_{res} (line 39).

4 Proof of Correctness

In this section we prove that our algorithm is linearizable. Let H be any history of the adaptive RMWable snapshot object.

▶ **Observation 1.** *From the algorithm, it is clear that for every integer $i \in [0 \ldots m-1]$:*

- *Whenever $lastUpdate[i] = (-, k, \bot)$, $k = 0$.*
- *Every successful SC operation on $lastUpdate[i]$ on line 45 changes $lastUpdate[i]$ from $(j, 0, \bot)$ to $(j+1, \bot, v)$ for some integer j and some non-$\bot$ value v such that between the matching LL operation on $lastUpdate[i]$ on line 42 and this successful SC operation on $lastUpdate[i]$, v is received from a* `Read()` *operation on $O[i]$ on line 44.*
- *Every successful SC operation on $lastUpdate[i]$ on line 49 changes $lastUpdate[i]$ from $(j, \bot, v)$ to (j, k, v) for some integer j, some positive integer k, and some non-$\bot$ value v such that between the matching LL operation on $lastUpdate[i]$ on line 42 and this successful SC operation on $lastUpdate[i]$, k is received from a* `FAI()` *operation on clk on line 48.*
- *Every successful SC operation on $lastUpdate[i]$ on line 49 sets $lastUpdate[i]$ to $(-, k, -)$ for some positive integer k that is greater than any previous successful SC operation on $lastUpdate[i]$ on line 49.*
- *Every successful SC operation on $lastUpdate[i]$ on line 56 changes $lastUpdate[i]$ from (j, k, v) to $(j, 0, \bot)$ for some integer j, some positive integer k, and some non-$\bot$ value v.*

We now assign every operation on $O[0 \ldots m-1]$ a *timestamp* that roughly approximates the order in which they occur:

▶ **Definition 2.** *For every integer $i \in [0 \ldots m-1]$, we assign every operation on $O[i]$ a timestamp as follows:*

- *For each* `Read()` *operation op_i on $O[i]$, let p be the process that performs op_i and v be the return value of op_i. If (i) op_i is performed when p executes line 44, (ii) p successfully performs an* `SC`$(-, \bot, v)$ *operation on $lastUpdate[i]$ when it next executes line 45, and (iii) the next successful SC operation on $lastUpdate[i]$ changes it to $(-, k, v)$ for some positive integer k, then the timestamp of op_i is this positive integer k.*
- *For each remaining operation op_i on $O[i]$, let op'_i be the earliest operation such that op'_i has a timestamp and op_i precedes op'_i. If op'_i exists, then the timestamp of op_i is the timestamp of op'_i; otherwise the timestamp of op_i is ∞.*

Thus by Observation 1 and Definition 2:

▶ **Observation 3.** *For every operation op_i on $O[i]$:*

- *If op_i precedes another operation op'_i on $O[i]$, then the timestamp of op_i cannot be greater than the timestamp of op'_i.*
- *If the timestamp of op_i is a positive integer k, then k is received from a* `FAI()` *operation on clk after op_i is performed on $O[i]$.*

We now define a completion H' of H, for which we will find a linearization.

▶ **Definition 4.** *Let H' be a completion of H such that:*

- *For each incomplete* `Invoke`$(i, \mathtt{op}_i)$ *operation op that has performed op_i on $O[i]$ on line 37 such that the timestamp of op_i is a positive integer (not ∞), op is completed with the same return value as op_i.*
- *All other incomplete operations are removed.*

The following lemma will help us prove that each `Invoke(`i, op_i`)` operation can be linearized between its invocation and response, because the timestamp of op_i is received during that interval. We will need it later in Definition 8, where we associate that timestamp with the `Invoke(`i, op_i`)` operation.

▶ **Lemma 5.** *For each* `Invoke(`i, $\texttt{op}_i$`)` *operation op in H', op has performed op_i on $O[i]$ and the timestamp of op_i is a positive integer k that is received from a* `FAI()` *operation on clk between $(inv(op), rsp(op))$ (or simply after $inv(op)$ if op is incomplete in H).*

Observation 6 below describes some important structural properties of the timestamps and the last bit stored in $lastScan[p][m]$.

▶ **Observation 6.** *From the algorithm, it is clear that for every process $p \in [0 \dots n-1]$:*

- *At any time t, there is a non-negative integer k and a value $v \in \{0, 1\}$ such that $lastScan[p][m] = (k, 0, 0, 0, v)$.*
- *Every successful SC operation on $lastScan[p][m]$ on line 3 changes $lastScan[p][m]$ from $(k, 0, 0, 0, 0)$ to $(k, 0, 0, 0, 1)$, for some non-negative integer k.*
- *Every successful SC operation on $lastScan[p][m]$ on line 10 changes $lastScan[p][m]$ from $(k, 0, 0, 0, 1)$ to $(k', 0, 0, 0, 0)$ for some positive integer $k' > k$ such that k' was previously received from a* `FAI()` *operation on clk on line 9.*
- *Only process p can set $lastScan[p][m]$ to $(-, 0, 0, 0, 1)$, and only on line 3.*

The following lemma will help us associate each `Click()` operation with a timestamp (see also Definition 8 below), which will then help us determine the linearization order.

▶ **Lemma 7.** *For every process $p \in [0 \dots n-1]$:*

1. *For every complete* `HelpScan(`p`)` *operation hs, there is a time t between $(inv(hs), rsp(hs))$ such that $lastScan[p][m]$ contains $(-, 0, 0, 0, 0)$ at time t.*
2. *For each* `Click()` *operation op in H' invoked by process p, there is a non-negative integer k such that op finds that $lastScan[p][m] = (k, 0, 0, 0, 0)$ on line 2 and then successfully changes $lastScan[p][m]$ to $(k, 0, 0, 0, 1)$ on line 3.*
3. *For each* `Click()` *operation op in H' invoked by process p, there is a positive integer k_{op} such that $lastScan[p][m] = (k_{op}, 0, 0, 0, 0)$ at time $rsp(op)$ and at some time $t \in (inv(op), rsp(op))$, some process performs a* `FAI()` *operation on clk on line 9 that returns k_{op}.*

We now assign every operation in H' an integer called its *timestamp*. These timestamps roughly approximate the order in which the operations occur, and so they are useful for constructing a linearization of H'.

▶ **Definition 8.** *We assign every operation in H' an integer* timestamp *as follows:*

- *For each* `Invoke(`i, $\texttt{op}_i$`)` *operation op in H', the timestamp of op in H' is the timestamp of op_i on $O[i]$. Note that by Lemma 5, this timestamp is a positive integer k such that k is received from a* `FAI()` *operation on clk between $(inv(op), rsp(op))$ (or simply after $inv(op)$ if op is incomplete in H).*
- *If op is a* `Click()` *operation by a process p, then by Lemma 7, there is a positive integer k_{op} such that at time $rsp(op)$, $lastScan[p][m] = (k_{op}, 0, 0, 0, 0)$. The timestamp of op is this positive integer k_{op}.*
- *If op is an* `Observe(`i`)` *operation by a process p, then the timestamp of op is the same as the timestamp of the last* `Click()` *operation by process p that precedes op; or 0 if no such* `Click()` *operation exists.*

Next, we define a linearization L of H'.

▶ **Definition 9.** *Let L be a linearization of H' such that:*

- *Each `Invoke(`i`, `op_i`)` operation with timestamp k is linearized ϵ infinitesimals before the time when a `FAI()` operation that returns k is applied on clk, where ϵ is the number of operations on $O[i]$ between op_i and the last operation on $O[i]$ with timestamp k.*
- *Each `Click()` operation with timestamp k is linearized at the time when a `FAI()` operation that returns k is applied on clk.*
- *Each `Observe(`i`)` operation is linearized at the end of its interval.*

▶ **Lemma 10.** *Each operation in H' has a unique, well-defined linearization point in L that is within its execution interval.*

The next lemma shows that the order of `Invoke(`i`, `op_i`)` operations in the linearization, L, is consistent with the order of op_i operations on $O[i]$.

▶ **Lemma 11.** *Let op be an `Invoke(`i`, `op_i`)` operation in H', and op' be an `Invoke(`i`, `op'_i`)` operation in H'. Then op precedes op' in L if and only if op_i precedes op'_i on $O[i]$.*

▶ **Observation 12.** *For every integer $i \in [0 \ldots m-1]$ and every integer $j \in [0 \ldots \Delta - 1]$, if some process p changes $R[j][i]$ from some value $(j', -, -)$ to some value (j'', k, v) at some time t, then:*

- *p does so on line 52.*
- *p found that $lastUpdate[i] = (j'', k, v)$ when it last executed line 42.*
- *$j' < j''$ and $j'' \bmod \Delta = j$.*
- *j'' is a positive integer, k is a positive integer and v is a non-$\perp$ value.*

The following two lemmas describe some structural properties of arrays R and $lastUpdate$, which will be useful for the linearization proof.

▶ **Lemma 13.** *For every integer $i \in [0 \ldots m-1]$, every positive integer j, every positive integer k, and every non-$\perp$ value v, if $lastUpdate[i]$ is set to (j, k, v) at some time t, let t' be the earliest time when a process executes line 52 after finding that $lastUpdate[i]$ contains (j, k, v) on line 42. Then at any time t_R, $R[j \bmod \Delta][i]$ is changed to (j, k, v) if and only if t' exists and $t' = t_R$.*

▶ **Lemma 14.** *For every integer $i \in [0 \ldots m-1]$, every positive integer k_r, every integer $j \in [0 \ldots \Delta - 1]$, every value j_r, and every value v_r, if $R[j][i]$ is set to (j_r, k_r, v_r) at some time t then j_r is a positive integer, $j_r \bmod \Delta = j$, $v_r \neq \perp$, and $lastUpdate[i] = (j_r, k_r, v_r)$ at time t.*

The next observation will describe how array $lastScan[0 \ldots n-1]$ can change.

▶ **Observation 15.** *From the algorithm, it is clear that for every process $p \in [0 \ldots n-1]$ and every integer $i \in [0 \ldots m-1]$:*

1. *$lastScan[p][i]$ can only be modified on line 34.*
2. *Let k be a positive integer, and t be the earliest time when $lastScan[p][i]$ contains $(k, -, -, -, -)$. Then at time t, $lastScan[p][i]$ contains $(k, maxKey, j_{left}, j_{right}, \perp)$ such that $maxKey \geq k$ and $dist(j_{left}, j_{right}) = \Delta - 1$. Furthermore, before time t, (i) there is a time when $lastScan[p][m]$ is set to $(k, 0, 0, 0, -)$, (ii) there is a time when $R[j_{right}][i]$ is set to $(-, maxKey, -)$, and (iii) there is no time when $lastScan[p][i]$ contains $(k', -, -, -, -)$ such that $k' \geq k$.*

3. *Let k be a non-negative integer and v be a non-$\bot$ value. Then $lastScan[p][i]$ cannot be changed from $(k, -, -, -, v)$ to $(k', -, -, -, -)$ for any non-negative integer $k' \leq k$.*
4. *At any time t, if $lastScan[p][i]$ is changed from $(k, maxKey, j_{left}, j_{right}, \bot)$ to $(k, maxKey', j'_{left}, j'_{right}, v')$, then (i) $maxKey' = maxKey$, (ii) $dist(j'_{left}, j'_{right}) = \lceil (dist(j_{left}, j_{right}) + 1)/2 \rceil - 1$, and (iii) $v' = \bot$ if and only if $dist(j'_{left}, j'_{right}) \neq 0$.*

We will now define successful `HelpObserve()` operations, which manage to update *lastScan*[*0* . . . *n* − *1*].

▶ **Definition 16.** *For each* `HelpObserve(`q, i`)` *operation ho, we say that ho is successful if and only if ho performs a successful SC operation on lastScan*$[q][i]$ *on line 34.*

The next lemma shows that each successful `HelpObserve()` operation corresponds to a `Click()` operation after which the helped `Observe()` operation can linearize.

▶ **Lemma 17.** *For every process $p \in [0 \ldots n-1]$ and every integer $i \in [0 \ldots m-1]$, if there is a successful* `HelpObserve(`p, i`)` *operation ho, then p invokes a* `Click()` *operation op in H such that ho executes line 34 after inv(op).*

▶ **Corollary 18.** *For every process $p \in [0 \ldots n-1]$ and every integer $i \in [0 \ldots m-1]$, lastScan$[p][i]$ can only be changed from its initial value $(0, 0, 0, 0, 0)$ after the invocation of a* `Click()` *operation by p.*

▶ **Lemma 19.** *For every process $p \in [0 \ldots n-1]$, every integer $i \in [0 \ldots m-1]$, every non-negative integer k, every non-$\bot$ value v, every non-negative integer j_{left} and every non-negative integer j_{right}, if $lastScan[p][m] = (k, 0, 0, 0, -)$ and $lastScan[p][i]$ is set to $(k, -, j_{left}, j_{right}, v)$ at some time t, then $R[j_{left}][i] = (-, k^*, v)$ at some time $t^* \leq t$, where k^* is the largest integer such that $k^* \leq k$ and there is a time when some entry of $R[0 \ldots \Delta - 1][i]$ contains $(-, k^*, -)$.*

The following technical lemma is critical for the linearizability proof; it helps us determine that `Observe()` operations follow the corresponding `Click()` operation.

▶ **Lemma 20.** *For every process $p \in [0 \ldots n-1]$ and every integer $i \in [0 \ldots m-1]$, if op is an* `Observe(`i`)` *operation with response v by p in H' and a positive integer k is the timestamp of op, then (i) $lastScan[p][i] = (k, -, -, -, v)$ at some time before rsp(op), and (ii) some entry of $R[0 \ldots \Delta - 1][i]$ contains $(-, k', -)$ for some integer $k' \geq k$ at some time before rsp(op).*

▶ **Lemma 21.** *Linearization L of H' respects the specification of the adaptive RMWable snapshot object.*

Proof. Suppose, for contradiction, that the linearization L of H' does not respect the specification of the adaptive RMWable snapshot object. Let *op* be the operation with the earliest linearization in L such that *op* violates the specification of the adaptive RMWable snapshot object, i.e., the return value of *op* differs from what *op* would have returned in a sequential history corresponding to L.

First assume that *op* is an `Invoke(`i, op_i`)` operation. From the algorithm, for each operation op'_i performed on $O[i]$, either op'_i is a `Read()` operation, or op'_i is performed on $O[i]$ by an `Invoke(`i, op'_i`)` operation in H on line 37. By Definition 2 and Definition 4, every `Invoke(`$i, -$`)` operation that is in H but not in H' does not perform any operation on $O[i]$ that precedes op_i. Furthermore, by Lemma 11, for each `Invoke(`i, op'_i`)` operation op' in H', op' precedes *op* in L if and only if op'_i precedes op_i on $O[i]$. Consequently, since `Read()` operations cannot change the state of $O[i]$, the return value of *op* cannot violate the specification of the adaptive RMWable snapshot object. This is a contradiction.

Now assume that op is an `Observe(`i`)` operation invoked by some process p. Let k_{op} be the timestamp of op.

First, consider the case where $k_{op} = 0$. Then by Definition 8, p does not invoke any `Click()` operation before op. Then according to the specification of the adaptive RMWable snapshot object, the `Observe(`i`)` operation op by p should return 0, the initial state of component i. Furthermore, by Corollary 18, $lastScan[p][i]$ always contains its initial value $(0, 0, 0, 0, 0)$ before $rsp(op)$.

So consider the first `HelpObserve(`p, i`)` operation ho called by op on line 13. Since $lastScan[p][i]$ always contains its initial value $(0, 0, 0, 0, 0)$ before $rsp(op)$, ho finds that $lastScan[p][i] = (0, 0, 0, 0, 0)$ on line 16. Then since there is no `Click()` operation by p before $rsp(op)$, by Observation 6 ho finds that $lastScan[p][m] = (0, 0, 0, 0, 0)$. So ho evaluates the conditional on line 18 as **false**, then evaluates the conditional on line 27 as **true**. Thus ho returns $v = 0 \neq \perp$ to op, and so op returns 0 on line 14 – contradicting that op violates the specification of the adaptive RMWable snapshot object.

So it remains to consider the case where $k_{op} > 0$. Then by Definition 8, p invokes `Click()` operation(s) before op, and the last `Click()` operation op' by p before op also has timestamp k_{op}. Thus by the definition of L, op' is linearized at the time that the `FAI()` operation that returns k_{op} is applied on clk.

Let v_{op} be the response value of op. Then by Lemma 20, (i) $lastScan[p][i] = (k_{op}, -, j_{left}, j_{right}, v_{op})$ at some time before $rsp(op)$, for some values j_{left} and j_{right}, and (ii) some entry of $R[0 \ldots \Delta-1][i]$ contains $(-, k', -)$ for some integer $k' \geq k_{op}$ at some time before $rsp(op)$. Let $t < rsp(op)$ be the time when $lastScan[p][i]$ is set to $(k_{op}, -, j_{left}, j_{right}, v_{op})$. Then by Observation 15(2), $lastScan[p][m]$ is set to $(k_{op}, 0, 0, 0, -)$ at some time $t_m < t$.

By Definition 8, since k_{op} is the timestamp of the `Click()` operation op' by p, $lastScan[p][m] = (k_{op}, 0, 0, 0, 0)$ at time $rsp(op')$. So by Observation 6, since op' is the last `Click()` operation by p before the `Observe(`i`)` operation op, $lastScan[p][m] = (k_{op}, 0, 0, 0, 0)$ between $(rsp(op'), rsp(op))$. Thus $t_m < rsp(op')$, and by Observation 6, $lastScan[p][m] = (k_{op}, 0, 0, 0, 0)$ between $(t_m, rsp(op))$. So since $t_m < t < rsp(op)$, $lastScan[p][m] = (k_{op}, 0, 0, 0, 0)$ at time t. Thus by Lemma 19 $R[j_{left}][i] = (-, k^*, v_{op})$ at some time $t^* \leq t$, where k^* is the largest integer such that $k^* \leq k_{op}$ and there is a time when some entry of $R[0 \ldots \Delta - 1][i]$ contains $(-, k^*, -)$.

Therefore, there is no integer $\hat{k}$ such that $k^* < \hat{k} \leq k_{op}$ and there is a time when some entry of $R[0 \ldots \Delta - 1][i]$ contains $(-, \hat{k}, -)$. Now recall that some entry of $R[0 \ldots \Delta - 1][i]$ contains $(-, k', -)$ for some integer $k' \geq k_{op}$ at some time before $rsp(op)$. So by Observation 1 and Lemma 13, from the algorithm it is clear that there is no integer $\hat{k}$ such that $k^* < \hat{k} \leq k_{op}$ and there is a time when $lastUpdate[i]$ contains $(-, \hat{k}, -)$. Thus by Definition 2 and Definition 8, there is no integer $\hat{k}$ such that $k^* < \hat{k} \leq k_{op}$ and some `Invoke(`$i, -$`)` operation in H' has timestamp $\hat{k}$. Now there are two cases: either $k^* = 0$, or $k^* > 0$.

First, consider the case where $k^* = 0$. Then, by the definition of L there are no `Invoke(`$i, -$`)` operations in H' linearized before the `Click()` operation op' by p. Thus according to the specification of the adaptive RMWable snapshot object, the `Observe(`i`)` operation op by p should return 0, the initial state of component i. Furthermore, since $k^* = 0$, by Observation 12 $R[j_{left}][i]$ still contains its initial value $(0, 0, 0)$ at time t^*, and so the `Observe(`i`)` operation op returns $v_{op} = 0$ – contradicting that op violates the specification of the adaptive RMWable snapshot object.

Now it remains to consider the case where $k^* > 0$. Let $\hat{t}^* \leq t^*$ be the time when $R[j_{left}][i]$ is set to $(-, k^*, v_{op})$. Then let j^* be an integer such that at time $\hat{t}^*$, $R[j_{left}][i]$ is set to (j^*, k^*, v_{op}). Then by Lemma 14, $lastUpdate[i] = (j^*, k^*, v_{op})$ at time $\hat{t}^*$. Thus by Observation 1, there is a process q such that (i) q performs a `Read()` operation op_i on $O[i]$ that returns v_{op} on line 44, (ii) q successfully performs an `SC(`$j^*, \perp, v_{op}$`)` operation on $lastUpdate[i]$ when it next executes line 45, and (iii) the next successful SC operation on $lastUpdate[i]$ changes it to (j^*, k^*, v_{op}). So by Definition 2, this `Read()` operation op_i on $O[i]$ has timestamp k^*.

Now recall that there is no integer $\hat{k}$ such that $k^* < \hat{k} \leq k_{op}$ and some `Invoke(`$i, -$`)` operation in H' has timestamp $\hat{k}$. So by the definition of L, every `Invoke(`$i, -$`)` operation in H' is linearized before the `Observe(`i`)` operation op if and only if its timestamp is at most k^*. Thus by Definition 2 and Definition 8, every `Invoke(`$i, -$`)` operation in H' that is linearized before op executes line 37 before the `Read()` operation op_i on $O[i]$ with timestamp k^*.

By Definition 2 and Definition 4, every `Invoke(`$i, -$`)` operation that is in H but not in H' does not perform any operation on $O[i]$ that precedes op_i. Furthermore, by Lemma 11, all `Invoke(`$i, -$`)` operations in H' are linearized by the order in which they execute line 37. Consequently, according to the specification of the adaptive RMWable snapshot object, the `Observe(`i`)` operation op by p should have the same response value as the `Read()` operation op_i on $O[i]$. Finally, recall that the response value of the `Read()` operation op_i on $O[i]$ is v_{op}, the response value of op – contradicting that op violates the specification of the adaptive RMWable snapshot object. Thus, we have shown that op is not an `Observe()` operation.

Since op is neither an `Invoke()` nor an `Observe()` operation, it must be a `Click()` operation. Thus the `Click()` operation op returns **done** on line 5 – contradicting that op violates the specification of the adaptive RMWable snapshot object. ◀

Consequently, this algorithm implements a linearizable adaptive RMWable snapshot object.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. of the ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.

2 Thomas Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1:6–16, 1990. doi:10.1109/71.80120.

3 James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Limited-use atomic snapshots with polylogarithmic step complexity. *J. of the ACM*, 62(1):1–22, 2015. doi:10.1145/2732263.

4 James Aspnes and Keren Censor-Hillel. Atomic snapshots in $O(\log^3 n)$ steps using randomized helping. In *Proc. of 27th DISC*, pages 254–268, 2013. doi:10.1007/978-3-642-41527-2_18.

5 Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In Friedhelm Meyer auf der Heide and Nir Shavit, editors, *Proc. of 20th SPAA*, pages 336–343, 2008. doi:10.1145/1378533.1378591.

6 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Efficient atomic snapshots using lattice agreement. In *Proc. of 6th WDAG*, pages 35–53, 1992.

7 Benyamin Bashari and Philipp Woelfel. An efficient adaptive partial snapshot implementation. In *Proc. of the 2021 ACM PODC*, pages 545–555, 2021. doi:10.1145/3465084.3467939.

8 Michiko Inoue and Wei Chen. Linear-time snapshot using multi-writer multi-reader registers. In *Proc. of the 8th WDAG*, pages 130–140, 1994. doi:10.1007/BFb0020429.

9 Prasad Jayanti. f-arrays: Implementation and applications. In *Proc. of 21st PODC*, pages 270–279, 2002. doi:10.1145/571825.571875.

10 Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. MemSnap: A fast adaptive snapshot algorithm for RMWable shared-memory. In *Proc. of 43rd PODC*, pages 25–35, 2024. doi:10.1145/3662158.3662820.

11 Prasad Jayanti and Srdjan Petrovic. Efficient and practical constructions of ll/sc variables. In *Proc. of 22nd PODC*, pages 285–294, 2003. doi:10.1145/872035.872078.

12 Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. on Comp.*, 30(2):438–456, 2000. doi:10.1137/S0097539797317299.

13 MirzaBaig, Danny Hendler, Alessia Milani, and Corentin Travers. Long-lived snapshots with polylogarithmic amortized step complexity. In *Proc. of the 2020 ACM PODC*, pages 31–40, 2020. doi:10.1145/3382734.3406005.

14 Robert Obryk. Write-and-f-array: implementation and an application. Master's thesis, Jagiellonian University, 2013.

15 Yaron Riany, Nir Shavit, and Dan Touitou. Towards a practical snapshot algorithm. *Theor. Comp. Sci.*, 269(1-2):163–201, 2001. doi:10.1016/S0304-3975(00)00412-6.

16 Yuanhao Wei, Naama Ben-David, Guy Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proc. of 26th PPOPP*, pages 31–46, 2021. doi:10.1145/3437801.3441602.

A Additional Proofs

This appendix contains some of the proofs omitted from Section 4.

In order to prove Lemma 5, we use the following statement, which describes how $\mathit{lastUpdate}[i]$ is affected by a complete `HelpUpdate(`i`)` operation.

▶ **Lemma 22.** *For each complete* `HelpUpdate(`i`)` *operation hu, there is a positive integer j such that:*

- *There are at least 6 successful SC operations on $\mathit{lastUpdate}[i]$ that occur between $(\mathit{inv}(\mathit{hu}), \mathit{rsp}(\mathit{hu}))$.*
- *Some process reads a non-$\perp$ value v from $O[i]$ on line 44 at some time $t_0 \in (\mathit{inv}(\mathit{hu}), \mathit{rsp}(\mathit{hu}))$, then successfully performs an* `SC(`$j, \perp, v$`)` *operation on $\mathit{lastUpdate}[i]$ when it next executes line 45 at some time $t_1 \in (t_0, \mathit{rsp}(\mathit{hu}))$.*
- *Some process (not necessarily distinct from the first) receives a positive integer k from a* `FAI()` *operation on clk on line 48 at some time $t_2 \in (t_1, \mathit{rsp}(\mathit{hu}))$, then performs the next successful SC operation on $\mathit{lastUpdate}[i]$ when it next executes line 49 at some time $t_3 \in (t_2, \mathit{rsp}(\mathit{hu}))$, which changes $\mathit{lastUpdate}[i]$ from $(j, \perp, v)$ to (j, k, v).*
- *Some process (not necessarily distinct from the first two) performs the next successful SC operation on $\mathit{lastUpdate}[i]$ on line 56 at some time $t_4 \in (t_3, \mathit{rsp}(\mathit{hu}))$, which changes $\mathit{lastUpdate}[i]$ from (j, k, v) to $(j, 0, \perp)$.*

Proof. From the algorithm, it is clear that in every outermost loop iteration of hu, hu performs an LL operation on $\mathit{lastUpdate}[i]$ on line 42, then performs an SC operation on $\mathit{lastUpdate}[i]$ (line 45, 49, or 56). So a successful SC operation on $\mathit{lastUpdate}[i]$ occurs within each loop iteration. Thus there are at least 6 successful SC operations on $\mathit{lastUpdate}[i]$ that occur between $(\mathit{inv}(\mathit{hu}), \mathit{rsp}(\mathit{hu}))$. Consequently, by Observation 1:

- There is a positive integer j and non-$\perp$ value v such that the second, third, or fourth successful SC operation on $\mathit{lastUpdate}[i]$ within $(\mathit{inv}(\mathit{hu}), \mathit{rsp}(\mathit{hu}))$ changes $\mathit{lastUpdate}[i]$ from $(j-1, 0, \perp)$ to $(j, \perp, v)$ at some time $t_1 \in (\mathit{inv}(\mathit{hu}), \mathit{rsp}(\mathit{hu}))$.
- The process that does this successful SC operation on $\mathit{lastUpdate}[i]$ at time t_1 reads v from $O[i]$ on line 44 at some time $t_0 \in (\mathit{inv}(\mathit{hu}), t_1)$.
- There is a positive integer k such that the next successful SC operation on $\mathit{lastUpdate}[i]$ within $(\mathit{inv}(\mathit{hu}), \mathit{rsp}(\mathit{hu}))$ changes $\mathit{lastUpdate}[i]$ from $(j, \perp, v)$ to (j, k, v) at some time $t_3 \in (t_1, \mathit{rsp}(\mathit{hu}))$.

- The process that does this successful SC operation on $\mathit{lastUpdate}[i]$ at time t_3 receives this positive integer k from a `FAI()` operation on clk on line 48 at some time $t_2 \in (t_1, t_3)$.
- The next successful SC operation on $\mathit{lastUpdate}[i]$ within $(\mathit{inv}(\mathit{hu}), \mathit{rsp}(\mathit{hu}))$ changes $\mathit{lastUpdate}[i]$ from (j, k, v) to $(j, 0, \bot)$ at some time $t_4 \in (t_3, \mathit{rsp}(\mathit{hu}))$.

Thus, since $\mathit{inv}(\mathit{hu}) < t_0 < t_1 < t_2 < t_3 < t_4 < \mathit{rsp}(\mathit{hu})$, the lemma holds. ◀

Proof of Lemma 5. First, consider the case where op is incomplete in H. Then by Definition 4, op has performed op_i on $O[i]$ and the timestamp of op_i is a positive integer k. So by Definition 2, there exists a `Read()` operation op'_i on $O[i]$, a process p that performs op'_i, and a value v returned by op'_i such that (i) op'_i is performed when p executes line 44, (ii) p successfully performs an `SC(`$-, \bot, v$`)` operation on $\mathit{lastUpdate}[i]$ when it next executes line 45, (iii) the next successful `SC()` operation on $\mathit{lastUpdate}[i]$ changes it to $(-, k, v)$, and (iv) op_i precedes op'_i on $O[i]$. Thus by Observation 1, this positive integer k is received from a `FAI()` operation on clk after op performs op_i on $O[i]$, which is clearly after $\mathit{inv}(op)$.

It now remains to the consider the case where op is complete in H. Thus op performs op_i on $O[i]$ on line 37, then calls `HelpUpdate(`i`)` on line 38. Since op is complete in H, this `HelpUpdate(`i`)` call completes before $\mathit{rsp}(op)$. So by Lemma 22, during this complete `HelpUpdate(`i`)` call:

- There are at least 6 successful SC operations on $\mathit{lastUpdate}[i]$
- Some process q receives non-$\bot$ value v from a `Read()` operation op'_i on $O[i]$ on line 44, then performs the second, third, or fourth successful SC operation on $\mathit{lastUpdate}[i]$ when it next executes line 45, changing it to $(-, \bot, v)$.
- The next successful SC operation on $\mathit{lastUpdate}[i]$ changes it to $(-, k', v)$ for some positive integer k' that was received from a `FAI()` operation on clk after the successful `SC(`$-, \bot, v$`)` operation on $\mathit{lastUpdate}[i]$ by q.

Thus by Definition 2, the timestamp of op'_i is this positive integer k'. As op_i precedes op'_i, by Definition 2 and Observation 3, the timestamp of op_i is a positive integer $k \leq k'$, which is returned from a `FAI()` operation on clk after op performs op_i on $O[i]$ on line 37. Hence, as $k' \geq k$ is received from a `FAI()` operation on clk during the complete `HelpUpdate(`i`)` call of op, k is received from a `FAI()` operation on clk during $(\mathit{inv}(op), \mathit{rsp}(op))$. ◀

Proof of Lemma 7. (1): Let hs be a complete `HelpScan(`p`)` operation. By Observation 6, there is a value $v \in \{0, 1\}$ such that hs finds that $\mathit{lastScan}[p][m]$ contains $(-, 0, 0, 0, v)$ on line 7. If $v = 0$, we are done. So suppose $v = 1$. Then hs evaluates the conditional on line 8 as **true**, gets a positive integer k from a `FAI()` operation on clk on line 9, and then performs an `SC(`$k, 0, 0, 0, 0$`)` operation on $\mathit{lastScan}[p][m]$ on line 10.

Let t_0 and t_1 be the times when hs executes lines 7 and 10 respectively. Note that $\mathit{inv}(hs) < t_0 < t_1 < \mathit{rsp}(hs)$. Then, since hs performs an LL operation on $\mathit{lastScan}[p][m]$ at time $t_0 > \mathit{inv}(hs)$, and an SC operation on $\mathit{lastScan}[p][m]$ at time $t_1 < \mathit{rsp}(hs)$, there must exist a successful SC operation on $\mathit{lastScan}[p][m]$ between $(\mathit{inv}(hs), \mathit{rsp}(hs))$. By Observation 6, every successful SC operation on $\mathit{lastScan}[p][m]$ changes it either from $(-, 0, 0, 0, 0)$ to $(-, 0, 0, 0, 1)$ or from $(-, 0, 0, 0, 1)$ to $(-, 0, 0, 0, 0)$. So there is a time t between $(\mathit{inv}(hs), \mathit{rsp}(hs))$ such that $\mathit{lastScan}[p]$ contains $(-, 0, 0, 0, 0)$ at time t. Thus we have proven (1).

(2): Initially, $\mathit{lastScan}[p][m] = (0, 0, 0, 0, 0)$. By Observation 6:

- Only process p can set $\mathit{lastScan}[p][m]$ to $(-, 0, 0, 0, 1)$, and only on line 3.
- Every successful SC operation on $\mathit{lastScan}[p][m]$ changes it either from $(-, 0, 0, 0, 0)$ to $(-, 0, 0, 0, 1)$ or from $(-, 0, 0, 0, 1)$ to $(-, 0, 0, 0, 0)$.

Thus only a `Click()` operation by process p can change $\mathit{lastScan}[p][m]$ from $(-, 0, 0, 0, 0)$, and only on line 3. Consequently, every complete `Click()` operation by process p in H:

- Finds that $\mathit{lastScan}[p][m]$ contains $(k, 0, 0, 0, 0)$ on line 2 for some non-negative integer k (Observation 6).
- Successfully changes $\mathit{lastScan}[p][m]$ to $(k, 0, 0, 0, 1)$ on line 3.
- Finishes with a HelpScan(p) call on line 4, which, since we have already proven (1), ensures that $\mathit{lastScan}[p][m]$ is changed to $(-, 0, 0, 0, 0)$ for the next Click() operation by p.

Finally, by Definition 4, every Click() operation in H' is complete in H. Thus we have proven (2).

(3): Let op be a Click() operation in H' that is invoked by process p. By Definition 4, every Click() operation in H' is complete in H, so op is complete in H. Since we have already proven (2), p successfully changes $\mathit{lastScan}[p][m]$ to $(-, 0, 0, 0, 1)$ on line 3 at some time $t_1 > \mathit{inv}(op)$. Then p calls HelpScan(p) on line 4, which, since we have already proven (1), ensures that some process q (not necessarily distinct from p) sets $\mathit{lastScan}[p][m]$ to $(k_{op}, 0, 0, 0, 0)$ for some value k_{op} at some time $t' < \mathit{rsp}(op)$.

By Observation 6, k_{op} is a positive integer, and at time t', q performs a successful SC(k_{op}, *0*, *0*, *0*, *0*) operation on $\mathit{lastScan}[p][m]$ on line 10 within a HelpScan() operation. Furthermore, since q performs a successful SC operation on $\mathit{lastScan}[p][m]$ on line 10, q must have performed the matching LL operation on $\mathit{lastScan}[p][m]$ on line 7 after the successful SC($-$, *0*, *0*, *0*, *1*) on $\mathit{lastScan}[p][m]$ by process p at time t_1. Thus q received k_{op} from a FAI() operation on clk on line 9 at some time $t \in (t_1, t')$. Then, since $t_1 > \mathit{inv}(op)$ and $t' < \mathit{rsp}(op)$, $t \in (\mathit{inv}(op), \mathit{rsp}(op))$.

Finally, from the algorithm it is clear that p does not execute line 3 after calling HelpScan(p) on line 4. So by Observation 6, $\mathit{lastScan}[p][m]$ cannot be changed again before $\mathit{rsp}(op)$, so $\mathit{lastScan}[p][m]$ still contains $(k_{op}, 0, 0, 0, 0)$ at time $\mathit{rsp}(op)$. Thus we have proven (3). ◀

Proof of Lemma 10. This is clearly true for all Observe(i) operations in H'.

By Lemma 7, for each Click() operation op in H' invoked by a process $p \in [0 \ldots n-1]$, there is a positive integer k_{op} such that $\mathit{lastScan}[p][m] = (k_{op}, 0, 0, 0, 0)$ at time $\mathit{rsp}(op)$ and at some time $t \in (\mathit{inv}(op), \mathit{rsp}(op))$, some process performs a FAI() operation on clk on line 9 that returns k_{op}. So by Definition 8, this positive integer k_{op} is the timestamp of op. Thus by Definition 9, op is linearized at the time $t \in (\mathit{inv}(op), \mathit{rsp}(op))$ when some process performs a FAI() operation on clk on line 9 that returns k_{op}. Consequently, every Click() operation in H' has a unique, well-defined linearization point in L that is within its execution interval.

Thus it remains to consider the Invoke() operations in H'. Let op be an Invoke(i, op_i) operation in H', and k be the timestamp of op in H'. Then by Definition 8, k is also the timestamp of op_i on $O[i]$. Then by Lemma 5, k is received from a FAI() operation on clk between $(\mathit{inv}(op), \mathit{rsp}(op))$. Then by Definition 9, there is a finite integer ϵ such that op is linearized ϵ infinitesimals before this FAI() operation on clk. Consequently, every Invoke() operation in H' has a unique, well-defined linearization point in L that is within its execution interval. ◀

Proof of Lemma 11. Suppose op_i precedes op'_i on $O[i]$. Then by Observation 3, the timestamp of op_i on $O[i]$ cannot be greater than the timestamp of op'_i on $O[i]$. So by Definition 9, the Invoke(i, op_i) operation op precedes the Invoke(i, op'_i) operation op' in L.

Thus if op_i precedes op'_i on $O[i]$, then op precedes op' in L. By symmetric arguments, if op'_i precedes op_i on $O[i]$, then op' precedes op in L. Consequently, op precedes op' in L if and only if op_i precedes op'_i on $O[i]$. ◀

Proof of Lemma 13. Suppose, for contradiction, that the lemma does not hold. Then let j be the smallest positive integer for which the lemma does not hold.

First, consider the case where t' does not exist, i.e., no process executes line 52 after finding that $\mathit{lastUpdate}[i]$ contains (j, k, v) on line 42. Then by Observation 12, no process ever sets $R[j \bmod \Delta][i]$ to (j, k, v) – contradicting that the lemma does not hold for j.

So it remains to consider the case where t' exists. Then t' is the earliest time when a process p executes line 52 after finding that $\mathit{lastUpdate}[i]$ contains (j, k, v) on line 42. By Observation 1:

- $\mathit{lastUpdate}[i]$ is never set to (j, k', v') for some non-$\perp$ values k' and v' such that $(k', v') \neq (k, v)$.
- $\mathit{lastUpdate}[i]$ is only set to (j, k, v) at time t (and so $t < t'$).
- Before $\mathit{lastUpdate}[i]$ is set to (j, k, v) at time t, $\mathit{lastUpdate}[i]$ never contains $(j', -, -)$ such that $j' > j$.

Thus by Observation 12, before time t', no process ever sets $R[j \bmod \Delta][i]$ to $(j', -, -)$ such that $j' \geq j$. So p finds that $R[j \bmod \Delta][i]$ contains $(j_{LL}, -, -)$ for some integer $j_{LL} < j$ on line 51. Thus at time t', p evaluates the conditional on line 52 as **true**, and performs an `SC(`j, k, v`)` operation on $R[j \bmod \Delta][i]$.

Since the lemma does not hold, this p must fail this `SC(`j, k, v`)` operation on $R[j \bmod \Delta][i]$ at time t'. Thus for some integer $\hat{j}$, at some time $\hat{t}$ that is between the time when p performs the matching LL operation on $R[j \bmod \Delta][i]$ on line 51 and time t', there is a successful `SC(`$\hat{j}, -, -$`)` operation on $R[j \bmod \Delta][i]$. So by Observation 12, $\hat{j}$ is a positive integer. Furthermore, recall that before time t', no process ever sets $R[j \bmod \Delta][i]$ to $(j', -, -)$ such that $j' \geq j$. Thus $\hat{j} < j$.

Now recall that j is the smallest positive integer for which the lemma does not hold. Thus the lemma holds for the positive integer $\hat{j} < j$. So $\hat{t}$ is the earliest time when a process executes line 52 after finding that $\mathit{lastUpdate}[i]$ contains $(\hat{j}, \hat{k}, \hat{v})$ on line 42 for some positive integer $\hat{k}$ and some non-$\perp$ value $\hat{v}$.

By Observation 1, $\mathit{lastUpdate}[i]$ can only be changed from $(\hat{j}, \hat{k}, \hat{v})$ on line 56. Thus from the algorithm, it is clear that at time $\hat{t}$, $\mathit{lastUpdate}[i]$ still contains $(\hat{j}, \hat{k}, \hat{v})$. Consequently, $\mathit{lastUpdate}[i]$ contains $(j, -, -)$ at time t and $(\hat{j}, -, -)$ at time $\hat{t}$ such that $t < \hat{t}$ and $j > \hat{j}$ – contradicting Observation 1. ◀

Proof of Lemma 14. Let q be the process that sets $R[j][i]$ to (j_r, k_r, v_r) at time t. By Observation 12, j_r is a positive integer, $j_r \bmod \Delta = j$, $v_r \neq \perp$, and q does so on line 52, after finding that $\mathit{lastUpdate}[i]$ contains (j_r, k_r, v_r) on line 42. Then, by Lemma 13, t is the earliest time when a process (namely q) executes line 52 after finding that $\mathit{lastUpdate}[i]$ contains (j_r, k_r, v_r) on line 42.

By Observation 1, $\mathit{lastUpdate}[i]$ can only be changed from (j_r, k_r, v_r) on line 56. Thus from the algorithm, it is clear that at time t, $\mathit{lastUpdate}[i]$ still contains (j_r, k_r, v_r). ◀

Proof of Lemma 17. Let ho be the successful `HelpObserve(`p, i`)` operation that executes line 34 earliest. Then let t be the time when ho executes line 34. By Observation 15(1) and Definition 16, $\mathit{lastScan}[p][i]$ can only be changed on line 34, within a successful `HelpObserve(`p, i`)` operation. So at time t, $\mathit{lastScan}[p][i]$ is changed from its initial value $(0, 0, 0, 0, 0)$. Thus by Observation 15(3), there is a positive integer $k > 0$ such that at time t, $\mathit{lastScan}[p][i]$ is changed to $(k, -, -, -, -)$. So by Observation 15(2), before time t, there is a time when $\mathit{lastScan}[p][m]$ is set to $(k, 0, 0, 0, -)$. By Observation 6, $\mathit{lastScan}[p][m]$ can only be changed from its initial value $(0, 0, 0, 0, 0)$ on line 3, within a `Click()` operation by process p. Thus p invokes a `Click()` operation op in H such that ho executes line 34 after $\mathit{inv}(op)$. ◀

▶ **Lemma 23.** *For every process $p \in [0 \ldots n-1]$, every integer $i \in [0 \ldots m-1]$, and every positive integer k, if lastScan$[p][i]$ is first set to $(k, -, -, -, -)$ at some time t_i, then clk returns k to a* `FAI()` *operation at some time $t < t_i$ and between (t, t_i), $R[0 \ldots \Delta-1][i]$ is modified at most $n+2$ times.*

The proof is omitted due to space restrictions.

Given any two integers j and j' in $[0 \ldots \Delta-1]$, we define $dist(j, j')$ to be $j' - j$, if $j' \geq j$, and $j' - j + \Delta$, otherwise. Note that if $j \neq j'$, then $dist(j, j') = \Delta - dist(j', j)$.

Proof of Lemma 19. First, consider the case where $k = 0$. Then by Observation 15(3), $lastScan[p][i]$ still contains its initial value $(0, 0, 0, 0, 0)$. Then, since $R[0][i]$ initially contains $(0, 0, 0)$, it is clear that the lemma holds.

So it remains to consider the case where k is a positive integer. Let t_i be the earliest time when when $lastScan[p][i] = (k, -, -, -, -)$. Since $lastScan[p][i]$ initially contains $(0, 0, 0, 0, 0)$, t_i exists and $t_i < t$. By Observation 15(2), at time t_i, $lastScan[p][i] = (k, maxKey, j'_{left}, j'_{right}, \perp)$, such that $maxKey \geq k$, $dist(j'_{left}, j'_{right}) = \Delta - 1$, and before time t_i, there is a time when $lastScan[p][m] = (k, 0, 0, 0, -)$ and a time when $R[j'_{right}][i] = (-, maxKey, -)$.

▶ **Subclaim 23.1.** *For each complete* `HelpObserve`(p, i) *operation ho such that $t_i < inv(ho) < rsp(ho) < t$, there is a successful* `HelpObserve`(p, i) *operation (not necessarily distinct from ho) that executes line 34 between $(inv(ho), rsp(ho))$.*

Proof. Consider *ho*:

- Since $t_i < inv(ho) < rsp(ho) < t$, by Observation 15 *ho* finds that $lastScan[p][i] = (k, -, -, -, \perp)$ on line 16.
- Since $lastScan[p][m] = (k, 0, 0, 0, -)$ at some time before time t_i and $lastScan[p][m] = (k, 0, 0, 0, -)$ at time $t > t_i$, by Observation 6, $lastScan[p][m]$ always contains $(k, 0, 0, 0, -)$ between (t_i, t). Thus since $t_i < inv(ho) < rsp(ho) < t$, *ho* finds that $lastScan[p][m] = (k, 0, 0, 0, -)$ on line 17.
- So *ho* evaluates the conditionals on lines 18 and 27 as **false**.
- Thus *ho* performs an SC operation on $lastScan[p][i]$ on line 34.

Consequently, by Definition 16 there exists a successful `HelpObserve`(p, i) operation (not necessarily distinct from *ho*) that executes line 34 between $(inv(ho), rsp(ho))$. ◀

Let t_k be the time when a `FAI()` operation on *clk* returns k. By Lemma 23, t_k exists and $t_k < t_i < t$.

The proofs of the following twwo claims are omitted due to space restrictions.

▶ **Subclaim 23.2.** *$R[0 \ldots \Delta-1][i]$ is modified $O(n \log \Delta/\Delta')$ times between (t_k, t).*

▶ **Subclaim 23.3.** *There is an integer $j^* \in [0 \ldots \Delta-1]$ such that:*

1. *$R[j'_{right}][i]$ always contains $(-, maxKey, -)$ between (t_i, t).*
2. *$R[j^*][i]$ always contains $(-, k^*, -)$ between (t_i, t).*
3. *For every integer $j \in [0 \ldots \Delta-1]$ such that $dist(j'_{left}, j) \leq dist(j'_{left}, j^*)$, at any time $\hat{t}$ such that $t_i \leq \hat{t} \leq t$, $R[j][i] = (-, \hat{k}, -)$ for some integer $\hat{k}$ such that either $\hat{k} < k$ or $\hat{k} > maxKey$.*
4. *For every integer $j \in [0 \ldots \Delta-1]$ such that $dist(j'_{left}, j) > dist(j'_{left}, j^*)$, at any time $\hat{t}$ such that $t_i \leq \hat{t} \leq t$, $R[j][i] = (-, \hat{k}, -)$ for some integer $\hat{k}$ such that $\hat{k} > k$ and $\hat{k} \leq maxKey$.*

Now consider each successful `HelpObserve`(p, i) operation ho' that sets $lastScan[p][i]$ to $(k, -, -, -, -)$ on line 34 after time t_i. Recall that t_i is the earliest time when $lastScan[p][i]$ contains $(k, -, -, -, -)$, t is the time when $lastScan[p][i]$ is set to $(k, -, j_{left}, j_{right}, v)$, and v is a non-$\perp$ value. So by Observation 15, from the algorithm it is clear that ho' executes line 16 after time t_i, and executes line 34 before or at time t. Furthermore, by Observation 15, there are integers j_1, j_2, j_3, and j_4 such that ho' changes $lastScan[p][i]$ from $(k, -, j_1, j_2, -)$ to $(k, -, j_3, j_4, -)$ on line 34. Thus from the algorithm it is clear that ho' evaluates the conditionals on lines 18 and 27 as **false**. Therefore ho':

- Finds that $lastScan[p][i] = (k, -, j_1, j_2, -)$ on line 16.
- Evaluates the conditional on line 18 as **false**.
- Finds that $R[j][i] = (-, k_r, -)$ on line 30, where j is an integer such that $dist(j_1, j) \leq dist(j_1, j_2)$.
- By Subclaim 23.3, evaluates the conditional on line 31 as **true** if and only if $dist(j'_{left}, j) > dist(j'_{left}, j^*)$.

Consequently $dist(j'_{left}, j_1) \leq dist(j'_{left}, j_3) \leq dist(j'_{left}, j^*) \leq dist(j'_{left}, j_4) \leq dist(j'_{left}, j_2) \leq dist(j'_{left}, j'_{right})$.

Finally, by Observation 15(1) and Definition 16, some successful `HelpObserve`(p, i) operation ho_t sets $lastScan[p][i]$ to $(k, -, j_{left}, j_{right}, v)$ on line 34 at time $t > t_i$. So, since $v \neq \perp$, by Observation 15(4), $j_{left} = j_{right} = j^*$. Thus ho_t finds that $R[j^*][i] = (-, -, v)$ on line 33 at some time between (t_i, t). Therefore by Subclaim 23.3, $R[j_{left}][i] = (-, k^*, v)$ at some time $t^* \leq t$, where k^* is the largest integer such that $k^* \leq k$ and there is a time when some entry of $R[0 \ldots \Delta - 1][i]$ contains $(-, k^*, -)$. ◀

Proof of Lemma 20. By Definition 8, k is also the timestamp of the last `Click()` operation op' by p that precedes op in H', and at $rsp(op')$, $lastScan[p][m] = (k, -, -, -, -)$. Note that by Observation 6, $lastScan[p][m]$ always contains $(k, -, -, -, -)$ between $(rsp(op'), rsp(op))$. Since the `Observe`(i) operation op returns v, op calls a `HelpObserve`(p, i) operation ho that returns v on line 13, and $v \neq \perp$.

Consider this `HelpObserve`(p, i) operation ho. On line 16, ho finds that $lastScan[p][i] = (k_i, -, -, -, v')$. Since $lastScan[p][m]$ always contains $(k, -, -, -, -)$ between $(rsp(op'), rsp(op))$, ho finds that $lastScan[p][m] = (k, -, -, -, -)$ on line 17. Then since ho returns $v \neq \perp$, from the algorithm it is clear that ho evaluates the conditional on line 18 as **false**, and so $k \leq k_i$. So by Observation 15(2), $k = k_i$. Finally, since ho does not return $\perp$, ho returns v' on line 27. Thus $v' = v$, and so ho found that $lastScan[p][i] = (k, -, -, -, v)$ on line 16.

Next, let k' be a value such that ho found that $lastScan[p][i] = (k, k', -, -, v)$ on line 16. Furthermore, let t be the earliest time when $lastScan[p][i]$ contains $(k, -, -, -, -)$. Then by Observation 15(4), $lastScan[p][i]$ contains $(k, k', -, -, -)$ at time t. Consequently, by Observation 15(2), $k' \geq k$ and before time t, there is a time when some entry of $R[0 \ldots \Delta - 1][i]$ is set to $(-, k', -)$. ◀

Hyperproperty-Preserving Register Specifications

Yoav Ben Shimon
Tel Aviv University, Israel

Ori Lahav
Tel Aviv University, Israel

Sharon Shoham
Tel Aviv University, Israel

Abstract

Reasoning about *hyperproperties* of concurrent implementations, such as the guarantees these implementations provide to randomized client programs, has been a long-standing challenge. Standard linearizability enables the use of *atomic specifications* for reasoning about standard properties, but not about hyperproperties. A stronger correctness criterion, called *strong linearizability*, enables such reasoning, but is rarely achievable, leaving various useful implementations with no means for reasoning about their hyperproperties. In this paper, we focus on registers and devise *non-atomic specifications* that capture a wide-range of well-studied register implementations and enable reasoning about their hyperproperties. First, we consider the class of *write strong-linearizable* implementations, a recently proposed useful weakening of strong linearizability, which allows more implementations, such as the well-studied single-writer ABD distributed implementation. We introduce a simple shared-memory register specification that can be used for reasoning about hyperproperties of programs that use write strongly-linearizable implementations. Second, we introduce a new linearizability class, which we call *decisive linearizability*, that is weaker than write strong-linearizability and includes multi-writer ABD, and develop a second shared-memory register specification for reasoning about hyperproperties of programs that use register implementations of this class. These results shed light on the hyperproperties guaranteed when simulating shared memory in a crash-resilient message-passing system.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Program specifications; Theory of computation → Distributed algorithms; Theory of computation → Concurrent algorithms

Keywords and phrases Hyperproperties, Concurrent objects, Distributed objects, Linearizability, Strong linearizability, Simulation

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.8

Related Version *Extended Version*: https://arxiv.org/abs/2408.11015 [21]

Funding This work is supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. [759102-SVIS] and 851811) and the Israel Science Foundation (grant number 2117/23 and 814/22).

1 Introduction

Linearizability [17] is a widely accepted correctness criterion for concurrent and distributed implementations of objects, allowing clients of an object to pretend that they use an atomic abstraction thereof, whose behaviors are much easier to understand [13]. The observational refinement between a linearizable implementation and its atomic specification is, however, restricted to reasoning about reachability of "bad" states. Dealing with more intricate properties, such as the ability of an adversary to control the probability distribution of the results of an object's methods, reveals that a linearizable implementation may manifest behaviors exceeding those permissible by the atomic specification [14]. In the terminology of [3],

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 8; pp. 8:1–8:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

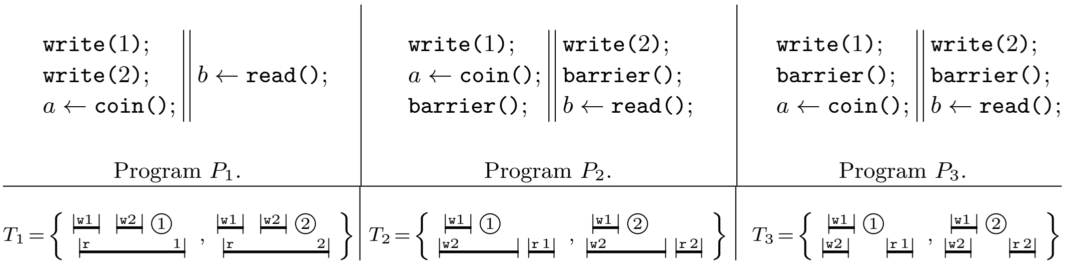
linearizability ensures preservation of safety properties but fails to maintain *hyperproperties*, which are properties of *sets* of executions, rather than individual executions. These properties allow one to express security guarantees, such as noninterference, as well as probability distributions on program outcomes [9].

The preservation of hyperproperties of concurrent implementations, a.k.a. *strong* observational refinement, necessitates a more strict connection between the implementation and its atomic specification, known as *strong linearizability* [14], which is equivalent to (a certain form of) forward simulation between the implementation and the atomic specification [3, 12]. Many implementations are, however, known to be *non*-strongly linearizable, leaving us with no means to reason about hyperproperties of programs that use these implementations by assuming simpler abstractions. In particular, the well-studied ABD implementation, which shows how shared memory can be simulated in a crash-tolerant message-passing system [2], is not strongly linearizable, and, in fact, a strongly linearizable implementation with similar guarantees does not exist [5].

Focusing on registers and observing that strong linearizability is hardly achievable, Hadzilacos et al. [15] recently proposed a weakening of strong linearizability, called *write strong-linearizability*, which captures more implementations. This includes single-writer ABD, and, in fact, as shown in [15], every linearizable implementation of a *single-writer* register. We are left, however, with substantial gaps: how should one reason about hyperproperties of programs that use write strongly-linearizable register implementations? and what can be said about existing non-strongly-linearizable implementations of *multi-writer* registers?

The current work aims to address these gaps. Inspired by Attiya and Enea [3], who propose to reason about hyperproperties of programs that use non-strongly-linearizable implementations by using simpler (albeit non-atomic) implementations related to them by strong observational refinement, we present a simple (but necessarily not atomic) specification of a shared multi-writer register, which we call WSR (for "Write Strong Register") that can be used for reasoning about hyperproperties of programs that use any write strongly-linearizable implementation (including single-writer ABD). To do so, we prove that every write strongly-linearizable implementation has a forward simulation to WSR, and utilize the correspondence between forward simulation and strong observational refinement (preservation of hyperproperties). Moreover, since write strong-linearizability is downward closed w.r.t. forward simulation and WSR is write strongly-linearizable, one can also prove write strong-linearizability for a given implementation by establishing a forward simulation to WSR, which may be more amenable to automatic/machine-assisted proofs than a direct proof. Drawing an analogy to complexity theory, we refer to WSR as a *complete* implementation for the class of write strongly-linearizable register implementations: WSR is write strongly-linearizable and every write strongly-linearizable implementation has a forward simulation to WSR.

As for multi-writer registers, we present a second specification of a shared register that is "complete" for a family of implementations that admit a weakening of write strong-linearizability, which we call *decisive linearizability.* In particular, we show that multi-writer ABD [19] belongs to this family. Thus, the complete implementation, which we call DR (for "Decisive Register"), enables reasoning about hyperproperties of programs that use ABD via a simpler *shared-memory specification.* (We also use DR to demonstrate that multi-writer ABD is decisively linearizable by showing a forward simulation to DR.) Intuitively speaking, unlike strong linearizability and write strong-linearizability, decisive linearizability gradually commits on the relative order of operations in the sequential history, rather than on the exact position of operations in that history.



```
write(1);    ||                 write(1);    || write(2);       write(1);    || write(2);
write(2);    || b ← read();     a ← coin();  || barrier();      barrier();   || barrier();
a ← coin();  ||                 barrier();   || b ← read();     a ← coin();  || b ← read();

        Program P1.                     Program P2.                     Program P3.
```

Figure 1 Client programs (upper part) and corresponding trace sets (lower part) that, if an adversary can generate them, violate the hyperproperty "$a = b$ with probability $\frac{1}{2}$".

Outline. The rest of this paper is structured as follows. In §2 we make the introductory discussion more concrete by outlining several examples. In §3 we provide the necessary preliminaries for our formal development. In §4 we introduce and study the notion of a complete implementation for a given linearizability class. In §5 we present the complete implementation for write strong-linearizability. In §6 we define decisive linearizability and present a complete implementation for this class. We discuss related work and conclude in §7. In Appendix A we provide proof sketches for several lemmas and theorems. The full version of this paper [21] provides full proofs.

2 Motivating Examples: A Tale of Four Registers

This section demonstrates certain intricacies arising when examining hyperproperties of client programs using (linearizable) implementations of concurrent registers. The specifications we develop in the next sections are based on the observations arising from these examples. We keep the discussion informal, deferring the formal treatment to the next sections.

Figure 1 (upper part) presents three programs, in which two threads read and write from a shared register, and invoke a method `coin()` that returns 1 or 2 uniformly at random. Programs P_2 and P_3 also employ a synchronization method `barrier()` that ensures the threads wait for each other before executing the rest of the code. Given that the underlying register implementation is linearizable, one can analyze standard properties of a single (finite) trace (e.g., the final values of the variables) of these programs by considering an *atomic* register (`ATR` in Fig. 2) [13]. In technical terms, one says that every linearizable implementation *observationally refines* the atomic register, and that the atomic register provides a *specification* (a.k.a. *reference implementation*) for any linearizable implementation.

However, as observed by Golab et al. [14], the atomic register cannot be used for analyzing properties of *sets* of program traces, a.k.a. *hyperproperties*, which cannot be deduced from a single program trace. For investigating hyperproperties, one considers the sets of program traces that can be generated by an adversary that controls the scheduling and the steps of the implementation. (By *program trace* we mean the sequence of actions performed by the client where the object's implementation internal actions are invisible.) Specifically, we consider the standard *strong* adversary that sees the whole execution so far and makes choices that depend on previous coin-toss results.

Atomic (ATR).

```
Method read()
    $out \leftarrow X$;
    return $out$;

Method write($v$)
    $X \leftarrow v$;
    return;
```

Double load (DLR).

```
Method read()
    $out_1 \leftarrow X$;
    $out_2 \leftarrow X$;
    if * then return $out_1$;
    else return $out_2$;

Method write($v$)
    $X \leftarrow v$;
    return;
```

Try-not-to-store (TNSR).

```
Method read()
    $out \leftarrow X$;
    return $out$;
Method write($v$)
    $a_1 \leftarrow X$;
    $a_2 \leftarrow X$;
    if * then
        if $a_1 \neq a_2$ then return;
    $X \leftarrow v$;
    return;
```

ABD implementation for N processes (ABD$_N$).

Shared Variables: A set *Broadcasts* of query/update messages and a mapping *Replies* from messages to their replies.

Local Variables: Process p stores the most recent value it observed, v_p, and its timestamp, ts_p. Timestamps are pairs $ts = \langle t, p \rangle$ with $t \in \mathbb{N}$ ordered lexicographically (assuming an arbitrary order on process id's). $\max\{\langle v_1, ts_1 \rangle, \dots, \langle v_n, ts_n \rangle\}$ retrieves the timestamped value $\langle v_i, ts_i \rangle$ with the maximum timestamp.

```
Method read()
    $\langle v, ts \rangle \leftarrow$query();
    update($v, ts$);
    return $v$;

Method write($v$)
    $\langle \_, \langle t, \_ \rangle \rangle \leftarrow$query();
    update($v, \langle t+1,$ my_process_id()$\rangle$);
    return;

Function update($v, ts$)
    broadcast $m =$ update($v, ts$);
    wait until $|Replies(m)| > N/2$;
    return;
```

```
Function query()
    broadcast $m =$ query;
    wait until $|Replies(m)| > N/2$;
    $Q \leftarrow$pick $Q \subseteq Replies(m)$ s.t. $|Q| > N/2$;
    return $\max Q$;

Background activity by process $p$:
when $m \in Broadcasts$ received
    if $m =$ query then
        reply $\langle v_p, ts_p \rangle$ to $m$;
    if $m =$ update($v, ts$) then
        $\langle v_p, ts_p \rangle \leftarrow \max\{\langle v, ts \rangle, \langle v_p, ts_p \rangle\}$;
        reply "ack" to $m$;
```

Figure 2 Four register implementations.

For instance, for the programs above, we may aim to verify that *under any adversarial scheduling the probability that $a = b$ at the end of execution is exactly $\frac{1}{2}$*, which indicates that the adversary cannot leak the coin-toss result from one thread to another.[1] With an atomic register, this property holds in all three programs. For instance, in programs P_1 and P_3, if the adversary performs the atomic `read()` before the coin is tossed, it cannot force a correlation between the coin and the read value; and by the time the coin is tossed, there is only one possible value that can be read.

Next, we demonstrate that this does not mean that other linearizable implementations guarantee this hyperproperty. To this end, we depict below each program in Fig. 1 a set of traces that forces $a = b$ with probability 1, and to show that the hyperproperty of a program is violated for certain implementations, we describe an adversary that generates this set.

We consider three linearizable register implementations, in addition to the atomic register (ATR) discussed above, presented in Figure 2: a "double load" implementation (DLR), a "try-not-to-store" implementation (TNSR), and the well-studied ABD implementation. Like ATR, DLR and TNSR are shared-memory implementations, using a single primitive (atomic) shared memory cell X initialized to 0 (all other variables are local). We refer to the accesses

[1] By adding conditional loops in the programs, one can correlate the probability that $a = b$ with the probability that the program diverges, and thus concentrate on asking whether an adversary can force non-termination, as considered in some previous work [15, 6].

to X as loads/stores, and to the methods of the register as reads/writes. In contrast, ABD is a register implementation in a crash-resilient message passing system, originally proposed to demonstrate that such a system can emulate a shared memory [2]. We present the multi-writer version of ABD from [19].

DLR. This implementation loads twice and non-deterministically picks which value to return (using **if** $*$). Using DLR, in P_1 the adversary can generate T_1 by ensuring this particular interleaving of the two threads, and moreover: execute the first load in the read method after 1 is stored to X, so that $out_1 = 1$; execute the second load after 2 is stored to X, so that $out_2 = 2$; and resolve the non-deterministic choice only after the coin is tossed ensuring that out_1 is returned if the coin result is 1, and out_2 is returned if the coin result is 2. (Recall that the adversary controls object-implementation-internal steps, including non-deterministic choices.) However, it is easy to see that for programs P_2 and P_3, the hyperproperty holds when DLR is used. Indeed, without a read concurrently executed with a write, DLR behaves just like ATR.

TNSR. This implementation tries to avoid some stores by recognizing that if the value is concurrently altered during a write operation, then that operation does not have to actually store as it may pretend it was overrun by the concurrent write. With this implementation, the hyperproperty holds for P_1. Indeed, without two concurrently executed writes, TNSR behaves just like ATR. However, using TNSR, in P_2 the adversary can generate T_2 by ensuring that the first load in `write(2)` reads 0 (the initial value), then execute `write(1)` atomically and have the second load in `write(2)` read 1. Then, if the coin result is 1, the adversary makes `write(2)` skip writing its value (it can do so since the two loaded values are not equal). Otherwise, if the coin result is 2, `write(2)` stores its value. Finally, it is easy to see that for P_3 the hyperproperty holds with TNSR. Indeed, after both threads reach the barrier, only one value can be returned by the read method, since at this point in the execution, both write methods are completed.

ABD. This implementation uses *timestamps* to order the written values (breaking ties using some predetermined order on the process identifiers). Each process maintains the most recent timestamped value it observed. For reading, a process broadcasts a query, waits for replies from a quorum (majority) of processes, and returns the value with the largest timestamp, but only after broadcasting this timestamped value and receiving acknowledgments from a quorum of processes. In turn, for writing value v a process broadcasts a query, waits for replies from a quorum of processes, broadcasts v with timestamp larger than all replies, and waits for a quorum of acknowledgments. Note that in ABD, processes are also constantly active as "servers": (*i*) replying to queries with their current timestamped values, and (*ii*) acknowledging new written values after (possibly) updating their current timestamped values.

Using ABD, the hyperproperty is violated for P_1 and P_2. For the violation we need to have at least three processes, two of them running the code of the program, and the others are used as servers that reply to messages and participate in quorums. ($\texttt{ABD}_2$ is degenerate since a quorum must consist of all processes.) Essentially, $\texttt{ABD}_{\geq 3}$ allows both the behaviors exposed by DLR and the behaviors exposed by TNSR. However, the actual adversaries for $\texttt{ABD}_{\geq 3}$ are more complicated than the ones for DLR and TNSR due to the absence of a global centralized memory cell that values are stored in and loaded from.

We describe adversaries that generate T_1 for P_1 and T_2 for P_2:

- For P_1, the adversary lets the reader invoke a query and lets the writer complete the execution of `write(1)` by choosing a quorum of processes that acknowledge the new value. Next, the adversary lets all the processes in the quorum reply to the query of

	ATR	DLR	TNSR	ABD$_{\geq 3}$
P_1	✓	✗	✓	✗
P_2	✓	✓	✗	✗
P_3	✓	✓	✓	✓

For each program and register implementation, ✓ indicates that the hyperproperty "$a = b$ with probability $\frac{1}{2}$" holds under any adversary, and ✗ indicates the hyperproperty is refuted by some adversary.

Figure 3 Summary of examples.

the reader reporting value 1. Then, the adversary lets the writer execute `write(2)`, again obtaining a quorum of processes that are aware of the new value, where this time the adversary picks a quorum that includes at least one process that is not part of the previous quorum, and therefore has not yet replied to the reader's query (this is where at least three processes are needed). This process also replies to the reader's query, but with value 2. At this time, the query message of the reader has pending replies from a quorum in which all replies include value 1, and from one additional process that is already aware of the more recent value 2. However, the adversary postpones the delivery of the replies until after the coin toss, at which time it picks the replies to match the coin value: if the coin value is 1, the replies from the homogeneous quorum where all replies include value 1 are delivered; otherwise the replies from the all but of one of the processes in the aforementioned quorum are delivered together with the reply of the additional process that includes value 2, thus forming a (heterogeneous) quorum whose most recent value is 2. Accordingly, the reader returns a value that is equal to the coin value.

- For P_2, the adversary starts by invoking a query during `write(2)` and making a quorum of processes send replies to the query (with the initial value) before `write(1)` is initiated in the left process. The adversary then lets the left process execute up to the barrier, at which point at least one reply with the value 1 is sent to the right process's query by a process that is aware of the left process's update. The adversary then performs the delivery of the replies to the query in `write(2)` according to the coin value. If the coin result is 2, the adversary delivers a quorum of replies that includes the reply sent when the left process reached the barrier, causing the right process to be aware of the most recent timestamp of the left process, such that the right process updates the value 2 with a larger timestamp. On the other hand, if the coin result is 1, the adversary delivers only the replies sent before `write(1)`, whose timestamp is outdated, causing the right process to choose a timestamp for the new value 2 that is at a tie with the timestamp attached by the left process to the value 1. Assuming the id of the left process has precedence, the tie is resolved to its timestamp, making 1 appear to be the most recent value. This determines the result of the subsequent read to be equal to the coin result.

Finally, the hyperproperty holds when ABD is used in P_3. To see this, suppose, w.l.o.g., that the timestamp assigned to 2 is larger than the one of 1. Then, after the two writes complete, in every quorum there is at least one process that knows about the value 2, and a reader that queries after this point can only read 2.

Figure 3 summarizes the above observations. In particular, the hyperproperty holds in P_3 for all four implementations. Nevertheless, as we show later in Example 6.2, it can be still violated by some linearizable implementations.

To capture differences between linearizable implementations, such as the ones shown in the above examples, [3] introduced *strong observational refinement* as a refinement relation between an implementation and a specification that preserves hyperproperties. Then, while ATR, DLR, TNSR, and ABD can be shown to be observationally equivalent (i.e., observationally refine each other), as we demonstrated above, they are not *strongly* observational equivalent. In particular, none of the relatively simple shared-memory implementations in Fig. 2 can be used as a specification of `ABD` when hyperproperties are considered, as `ABD` is not a strong observational refinement of any of them. (This is unfortunate, since, as we have seen, reasoning about the sets of program traces generated when `ABD` is used is much more involved than with the other implementations.) We also note that each of the implementations admits a different linearizability criterion: `ATR` is strongly linearizable [14], `DLR` is write strongly-linearizable [15], while `TNSR` and `ABD` are neither.

In the rest of the paper we propose hyperproperty-preserving specifications for classes of linearizable register implementations, including `ABD`. Such specifications can drastically simplify verification of hyperproperties of client programs using these implementations, a task which is typically challenging, especially when complex implementations are considered, since it requires reasoning about *all* possible adversaries.

3 Preliminaries

We start with general notations, continue to our modeling of objects, implementations, and programs (§3.1), and finally recap the formal notions of preservation of hyperproperties via strong observational refinement (§3.2).

Sequences. For a finite alphabet Σ, we denote by Σ^* the set of all (finite) sequences over Σ. The length of a sequence s is denoted by $|s|$. We write $s[k]$ for the symbol at position $1 \leq k \leq |s|$ in s. We write $\sigma \in s$ if $s[k] = \sigma$ for some $1 \leq k \leq |s|$. We use "$\cdot$" for the concatenation of sequences. We often identify symbols with sequences of length 1 or their singletons (e.g., in expressions like $s \cdot \sigma$). The *restriction* of a sequence s w.r.t. a set Γ, denoted by $s|_\Gamma$, is the longest subsequence of s that consists only of symbols in Γ. This notation is extended to sets by $S|_\Gamma \triangleq \{s|_\Gamma \mid s \in S\}$. We write $s_1 \preceq_{\mathsf{S}} s_2$ when s_1 is a subsequence of s_2, and $s_1 \preceq_{\mathsf{P}} s_2$ when s_1 is a prefix of s_2.

Labeled Transition Systems. A *labeled transition system* (LTS, for short) is a tuple $A = \langle Q, \Sigma, q_0, T\rangle$, where Q is a set of *states*, Σ is a (possibly infinite) alphabet (whose elements are called *transition labels*), $q_0 \in Q$ is an *initial state*, and $T \subseteq Q \times \Sigma \times Q$ is a set of *transitions*. We denote by $A.\mathtt{Q}$, $A.\mathbf{\Sigma}$, $A.\mathtt{q}_0$, and $A.\mathtt{T}$ the components of an LTS A. We write $\xrightarrow{\sigma}_A$ for the relation $\{\langle q, q'\rangle \mid \langle q, \sigma, q'\rangle \in A.\mathtt{T}\}$. An *execution* e of A is a (possibly empty) finite sequence of transitions in $A.\mathtt{T}$ such that the first transition starts in q_0 and each other transition continues from the target of the previous transition. An execution e induces a *trace* $\rho \in A.\mathbf{\Sigma}^*$, where $\rho[i]$ is given by the label of $e[i]$ for every $1 \leq i \leq |e|$. We denote by $\mathsf{E}(A)$ and $\mathsf{traces}(A)$ the set of all executions of A and the set of all traces induced by executions of A (respectively). Note that we only consider finite executions and traces.

Forward Simulations. Given LTSs A and $A^\#$ and a set $\Gamma \subseteq A.\mathbf{\Sigma} \cap A^\#.\mathbf{\Sigma}$, a relation $R \subseteq A.\mathtt{Q} \times A^\#.\mathtt{Q}$ is a Γ-*forward simulation* from A to $A^\#$ if

(i) $\langle A.\mathtt{q}_0, A^\#.\mathtt{q}_0\rangle \in R$; and

(ii) if $q \xrightarrow{\sigma}_A q'$ and $\langle q, q^\#\rangle \in R$, then there exist ${q^\#}' \in A^\#.\mathtt{Q}$ and $\rho \in A^\#.\mathbf{\Sigma}^*$ such that $q^\# \xrightarrow{\rho[1]}_{A^\#} \dots \xrightarrow{\rho[|\rho|]}_{A^\#} {q^\#}'$, $\rho|_\Gamma = \sigma|_\Gamma$, and $\langle q', {q^\#}'\rangle \in R$.

We write $A \sqsubseteq^\Gamma_{\mathsf{F}} A^\#$ when such relation exists.

3.1 Objects, Implementations, and Programs

We review standard notions that are needed for our formal results. We assume a set Tid of thread identifiers and an infinite set Id of action identifiers.

Objects. An *object* is a pair $O = \langle \mathsf{M}, \mathsf{Val} \rangle$, where M is a set of method names and Val is a set of values. An object O is associated with actions divided into *invocations* $i = \mathtt{inv}\langle m, v, p, k \rangle \in \mathsf{I}(O)$ and *responses* $r = \mathtt{res}\langle m, v, p, k \rangle \in \mathsf{R}(O)$, where $m \in \mathsf{M}$, $v \in \mathsf{Val} \cup \{\bot\}$, $p \in \mathsf{Tid}$, and $k \in \mathsf{Id}$. We let $\mathsf{IR}(O) \triangleq \mathsf{I}(O) \cup \mathsf{R}(O)$.

Histories. A *history* h of an object O is a finite sequence over $\mathsf{IR}(O)$. A history h is *sequential* if it alternates between invocations and responses (starting with an invocation), such that every consecutive i, r in h have the same method and thread identifiers, and a unique action identifier across h. A history h is *well-formed* if its restriction to actions of each $p \in \mathsf{Tid}$, denoted by $h|_p$, is sequential. An invocation $i \in h$ is *pending* if there is no response in h with the same thread and action identifiers. Otherwise, i is *complete*. These notions are also applied on *operations* o, which are either single invocations $o = i$ or pairs of matching invocation and response $o = \langle i, r \rangle$. We let $\mathsf{completed}(h)$ denote the subsequence of h consisting of actions that are a part of completed operations.

Real-time Order. The *real time order* induced by a well-formed history h, denoted by $<_h$, is the partial order on operations defined by $o_1 <_h o_2$ iff o_1's response appears in h before o_2's invocation.

Specifications. A *specification* of O is a prefix-closed set of sequential histories of O.

Registers. A register object is given by $\mathsf{Reg} = \langle \{\mathtt{read}, \mathtt{write}\}, \mathbb{N} \rangle$. Its specification, denoted by $\mathsf{Spec}_{\mathsf{Reg}}$, is defined as usual, assuming that 0 is the initial register value.

Object Implementations. We assume a set IInt of labels for implementation internal actions and define an *implementation* I of an object O to be an LTS over the alphabet $\mathsf{IR}(O) \cup \mathsf{IInt}$. We assume that the history induced by every execution e of I, denoted by $\mathsf{h}(e)$, is a well-formed history. The pseudo-code presented in specific implementations in the paper is easily translatable to formal LTSs, whose executions represent executions generated by the methods' code when they are repeatedly and concurrently invoked with arbitrary arguments.

Client Programs. We assume a set PInt of labels for client internal actions (disjoint from IInt) and define a client program P for an object O as an LTS over the alphabet $\mathsf{IR}(O) \cup \mathsf{PInt}$. A program P and implementation I are *linked* by taking "interface parallel composition", denoted by $P[I]$. The resulting LTS interleaves the steps of P and I while forcing the two LTSs to synchronize on labels from $\mathsf{IR}(O)$. The defining property of $P[I]$ is given by:

▶ **Proposition 3.1.** $\rho \in \mathsf{traces}(P[I])$ *iff* $\rho|_{I.\mathbf{\Sigma}} \in \mathsf{traces}(I)$ *and* $\rho|_{P.\mathbf{\Sigma}} \in \mathsf{traces}(P)$.

3.2 Hyperproperties Preservation via Strong Observational Refinement

A *hyperproperty* ϕ of a program P is a set of sets of the program's traces (i.e., $\phi \subseteq \mathcal{P}(\mathsf{traces}(P))$). Such sets can capture probabilistic requirements, such as the one informally described in §2, via suitable encodings of traces [9].

The hyperproperties that are satisfied by an object implementation, and accordingly, strong observational refinement between implementations, are defined using deterministic schedulers, which formalize the notion of a strong adversary [3].

Schedulers. Given a program P and an implementation I, a *scheduler* is a function $S : \mathsf{E}(P[I]) \to \mathcal{P}(P[I].\mathtt{T})$. An execution $e \in \mathsf{E}(P[I])$ is *consistent with* S if $e[j] \in S(e[1] \cdots e[j-1])$ for every $1 \leq j \leq |e|$. We denote by $\mathsf{E}(P[I], S)$ the set of executions of $P[I]$ that are consistent with S, and by $\mathsf{traces}(P[I], S)$ the traces of executions in $\mathsf{E}(P[I], S)$. A scheduler is *deterministic* if for every $e \in \mathsf{E}(P[I])$, either $|S(e)| \leq 1$ or all transitions in $S(e)$ are labeled by actions in PInt.

▶ Remark 3.2. Attiya and Enea [3] restricted their attention to *step-deterministic* implementations in which a trace uniquely determines an execution (which includes the intermediate states along the trace). We avoid this technical restriction, and thus use executions instead of traces in the definitions of schedulers, as well as of linearizability criteria below. In particular, we define schedulers as functions from executions to sets of transitions instead of functions from traces to sets of labels. For step-deterministic implementations our definitions coincide with those of [3].

Hyperproperty Satisfaction. An implementation I *satisfies a hyperproperty* ϕ of P, denoted by $I \models_P \phi$, if $\mathsf{traces}(P[I], S)|_{P.\mathbf{\Sigma}} \in \phi$ for every deterministic scheduler S.

▶ **Example 3.3.** For the client program P_2 (represented as an LTS) and the set of traces T_2 from Fig. 1, we have that $\mathtt{DLR} \models_{P_2} \mathcal{P}(\mathsf{traces}(P_2)) \setminus \{T_2\}$. This is because, as discussed in §2, there exists no scheduler S such that $\mathsf{traces}(P_2[\mathtt{DLR}], S)|_{P.\mathbf{\Sigma}} = T_2$. ⌟

Strong Observational Refinement. An implementation I *strongly observationally refines* an implementation $I^\#$, denoted by $I \leq_{\mathsf{s}} I^\#$, if $I^\# \models_P \phi \implies I \models_P \phi$ for every program P and hyperproperty ϕ of P. The following alternative characterization follows from the definition.

▶ **Lemma 3.4.** *$I \leq_{\mathsf{s}} I^\#$ iff for every program P and deterministic scheduler S, there exists a deterministic scheduler $S^\#$ such that* $\mathsf{traces}(P[I], S)|_{P.\mathbf{\Sigma}} = \mathsf{traces}(P[I^\#], S^\#)|_{P.\mathbf{\Sigma}}$.

Attiya and Enea [3, Theorem 8] show that $\mathsf{IR}(O)$-forward simulation between implementations is equivalent to strong observational refinement. (Their result applies to finite traces as we consider here; see [12] for a discussion on infinite traces.) We adapt this result to our setting. In the sequel, for implementations I and $I^\#$ of an object O, we write $I \sqsubseteq_{\mathsf{F}} I^\#$ for $I \sqsubseteq_{\mathsf{F}}^{\mathsf{IR}(O)} I^\#$.

▶ **Theorem 3.5.** *$I \leq_{\mathsf{s}} I^\#$ iff $I \sqsubseteq_{\mathsf{F}} I^\#$.*

▶ **Example 3.6.** It is easy to show that $\mathtt{ATR} \sqsubseteq_{\mathsf{F}} \mathtt{DLR}$, and we obtain that $\mathtt{ATR} \leq_{\mathsf{s}} \mathtt{DLR}$. Thus, $\mathtt{ATR} \models_{P_2} \mathcal{P}(\mathsf{traces}(P_2)) \setminus \{T_2\}$ follows from $\mathtt{DLR} \models_{P_2} \mathcal{P}(\mathsf{traces}(P_2)) \setminus \{T_2\}$. In addition, since $\mathtt{DLR} \not\leq_{\mathsf{s}} \mathtt{ATR}$ (see §2), we have $\mathtt{DLR} \not\sqsubseteq_{\mathsf{F}} \mathtt{ATR}$. Indeed, if a concurrent write is about to change the value of X after a read of $\mathtt{DLR}$ performs its first load, $\mathtt{ATR}$ has no matching action: if it performs its (single) load it will not be able to return the right value in case $\mathtt{DLR}$ returns the value read in the second load; and similarly, if it waits, it will fail to return the same value if $\mathtt{DLR}$ returns the value of the first load. ⌟

4 Complete Implementations for Linearizability Classes

Knowing that a given implementation is a member of a certain linearizability class is only useful if it enables reasoning about programs that use that implementation without understanding the implementation itself. For hyperproperties, such reasoning is made possible if the implementation is known to strongly observationally refine a simpler implementation, in which case the latter can be used instead of the actual implementation in the analysis. To standardize the relation between linearizability classes and strong observational refinement, we propose a definition of *hard* and *complete* implementations in analogy to hardness and completeness w.r.t. complexity classes, where instead of reductions, we use simulations, which ensure strong observational refinement:

▶ **Definition 4.1.** Let $\mathcal{I}$ be a class of implementations of an object O that is downward closed w.r.t. forward simulation (i.e., $I \in \mathcal{I}$ whenever $I' \in \mathcal{I}$ and $I \sqsubseteq_{\mathsf{F}} I'$). An implementation $I^\#$ of O is *$\mathcal{I}$-hard* if $I \sqsubseteq_{\mathsf{F}} I^\#$ for every $I \in \mathcal{I}$. It is *$\mathcal{I}$-complete* (or *complete for $\mathcal{I}$*) if we also have $I^\# \in \mathcal{I}$.

In addition to allowing reasoning about hyperproperties of implementations in $\mathcal{I}$, an $\mathcal{I}$-complete implementation $I^{\#}$ also provides a sound and complete method to establish the membership of an implementation I in $\mathcal{I}$ by showing that $I \sqsubseteq_{\mathsf{F}} I^{\#}$.

In the following we take $\mathcal{I}$ to be the set of implementations of some object that satisfy certain linearizability criteria.

Linearizability. Consider first standard linearizability [17, 20]:

▶ **Definition 4.2.** A history s of an object O is a *linearization* of a history h of O, denoted by $h \sqsubseteq s$, if there exists a sequence of responses $\bar{r}$ for some of the pending invocations in h such that the following hold for $h' = \mathsf{completed}(h \cdot \bar{r})$:

(i) $h'|_p = s|_p$ for every $p \in \mathsf{Tid}$; and

(ii) $<_{h'} \subseteq <_s$.

A history h of O is *linearizable* w.r.t. a specification *Spec* of O if it has a linearization $s \in \mathit{Spec}$. An implementation I of O is *linearizable* w.r.t. *Spec* if $\mathsf{h}(e)$ is linearizable w.r.t. *Spec* for every $e \in \mathsf{E}(I)$.

▶ **Proposition 4.3.** *The class of linearizable implementations of an abject O w.r.t. a specification Spec is downward closed w.r.t. forward simulation, and there exists a complete implementation for it.*

Proof (sketch). Downward closedness follows from the fact that $I \sqsubseteq_{\mathsf{F}} I'$ implies that $\{\mathsf{h}(e) \mid e \in \mathsf{E}(I)\} \subseteq \{\mathsf{h}(e) \mid e \in \mathsf{E}(I')\}$. A complete implementation is the implementation that tracks in its internal state the history h generated so far. When executing an invocation or response, the action is added in the end of the current history. But, while invocations are always enabled, a response r is only enabled when $h \cdot r$ is linearizable w.r.t. *Spec*. ◀

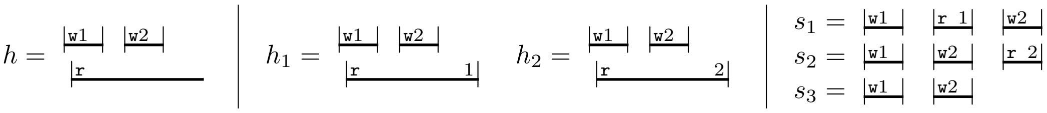
The (theoretical) construction in the above proof provides us with a complete implementation, which may help in streamlining and mechanizing linearizability arguments as forward simulations (e.g., [18] utilized such implementation). However, since it directly encodes the definition of the class, it is unhelpful for reasoning about hyperproperties of implementations. Thus, for the stronger classes considered below we are interested in identifying simple complete implementations that are not based on history tracking.

Strong linearizability. Golab et al. [14] proposed a strengthening of linearizability, called *strong linearizability*, and showed that it is necessary and sufficient for reasoning on probability distributions of outcomes that a strong adversary can generate. Roughly speaking, while linearizability allows one to choose the linearization order "after the fact" in view of the whole execution, strong linearizability requires the linearization of implementation histories into specification histories to be done online in a prefix-preserving manner, that is, by continuously adding operations at the end of the linearized history.

▶ **Definition 4.4.** A *linearization mapping* for an implementation I of an object O w.r.t. a specification *Spec* of O is a function $L : \mathsf{E}(I) \to \mathit{Spec}$ such that $\mathsf{h}(e) \sqsubseteq L(e)$ for every $e \in \mathsf{E}(I)$. An implementation I of O is *strongly linearizable* w.r.t. a specification *Spec* of O if there is a linearization mapping L for I w.r.t. *Spec* such that $L(e_1) \preceq_{\mathsf{P}} L(e_2)$ whenever $e_1 \preceq_{\mathsf{P}} e_2$.

Since we aim to also capture non-deterministic implementations (and do not assume step-determinism), our linearizations apply on *executions* rather than traces (see also Remark 3.2).

▶ **Example 4.5.** From the register implementations presented in §2, only ATR is strongly linearizable. We use the histories associated with the set T_1 from Fig. 1 to show that DLR and ABD are not strongly linearizable. Consider the following history h, its two possible extensions h_1 and h_2, and its possible linearizations s_1, s_2, s_3:



Unlike ATR (and TNSR), both DLR and ABD have a *single* execution e that induces h and can be extended into two alternative executions that induce h_1 and h_2. Then, $L(e)$ can be s_1, s_2, or s_3, but any choice at this stage is doomed to fail:

(i) s_1 fails if the execution continues to generate h_2;

(ii) s_2 fails if the execution continues to generate h_1; and

(iii) s_3 fails if the execution continues to generate h_1 since we are only allowed to extend the current linearization by adding operations at its end. The history of the common prefix of the traces in T_2 from Fig. 1 can be similarly used to show that TNSR is not strongly linearizable. ⌟

Attiya and Enea [3] show that the class of strongly linearizable implementations is downward closed w.r.t. forward simulation, and that every strongly linearizable implementation strongly observationally refines the atomic implementation (e.g., ATR for registers). Together with Thm. 3.5, this result is restated as follows:

▶ **Theorem 4.6.** *The atomic implementation for specification Spec of an object O is complete for the class of strongly linearizable implementations of O w.r.t. Spec.*

Additional linearizability classes. We observe that downward-closedness w.r.t. simulation, as well as the existence of a complete implementation, generalize to a range of linearizability classes beyond linearizability and strong linearizability mentioned above. These linearizability classes are parameterized by a preorder that must hold between the linearizations of an execution and its extensions. Formally, given a preorder R (i.e., reflexive and transitive relation) on sequences, the class $\mathcal{I}_R(O, \mathit{Spec})$ consists of all implementations I of O for which there exists a linearization mapping $L : \mathsf{E}(I) \to \mathit{Spec}$ such that $\langle L(e_1), L(e_2)\rangle \in R$ whenever $e_1 \preceq_{\mathsf{P}} e_2$. The class of all linearizable implementations of O w.r.t. *Spec* is obtained by taking $R = \mathit{Spec} \times \mathit{Spec}$, whereas for all strongly linearizable implementations we take $R =\preceq_{\mathsf{P}}$. Other classes defined in the rest of this paper are also instances of this definition.

▶ **Lemma 4.7.** *For every preorder R on sequences, the class $\mathcal{I}_R(O, \mathit{Spec})$ is downward closed w.r.t. forward simulation, and there exists a complete implementation for it.*

The complete implementation for $\mathcal{I}_R(O, \mathit{Spec})$ is constructed similarly to the one in the proof of Prop. 4.3 (which is a special case), except that here the state also tracks a linearization of the history so far, and ensures in each transition that the linearizations in the pre-state and post-state are related by R.

Similarly to the construction in Prop. 4.3, the generic construction in Lemma 4.7 is not helpful for reasoning about hyperproperties. In contrast, Thm. 4.6 proposes a simple and useful complete implementation for strong linearizability. In the remainder of the paper we seek useful complete implementations for other linearizability classes of interest.

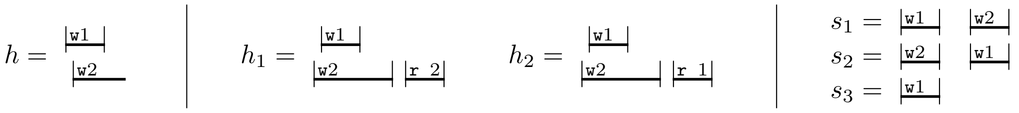
5 Complete Implementation for Write Strong Linearizability

Focusing on registers and identifying that useful register implementations are not strongly linearizable, Hadzilacos et al. [15] have recently proposed a weakening of strong linearizability, called *write strong-linearizability*, and showed that every linearizable *single writer* register implementation, including single-writer ABD, is write strongly-linearizable. However, they do not provide a specification for write strong-linearizability that plays the role that the atomic register implementation plays for strong linearizability.

Write strong-linearizability weakens the prefix-preservation requirement of strong linearizability by applying it only to writes, thus allowing reads to be linearized offline, and freely "move around" when more operations are added. For the formal definition, we let $s|_{\texttt{write}}$ denote the restriction of $s \in \mathsf{Spec}_{\mathsf{Reg}}$ to write operations.

▶ **Definition 5.1.** Let I be a register implementation. A linearization mapping $L : \mathsf{E}(I) \to \mathsf{Spec}_{\mathsf{Reg}}$ is *write strong* if $L(e_1)|_{\texttt{write}} \preceq_{\mathsf{P}} L(e_2)|_{\texttt{write}}$ whenever $e_1 \preceq_{\mathsf{P}} e_2$. We say that I is *write strongly-linearizable* if there exists a write strong linearization mapping $L : \mathsf{E}(I) \to \mathsf{Spec}_{\mathsf{Reg}}$.

▶ **Example 5.2.** From the implementations in §2, ATR and DLR are write strongly-linearizable. (For DLR, for h from Example 4.5, we can pick s_3, and later on, when the read returns, pick either s_1, by adding a read in the middle, or s_2 according to the returned value.) We use the histories associated with the set T_2 from Fig. 1 to show that TNSR and ABD are not write strongly-linearizable. (For ABD this also follows from the general result in [8].) Consider the following history h, its two possible extensions h_1 and h_2, and its possible linearizations s_1, s_2, s_3:



Unlike ATR and DLR, both TNSR and ABD have a *single* execution e that induces h and can be extended into two alternative executions that induce h_1 or h_2. Then, $L(e)$ can be s_1, s_2, or s_3, but any choice at this stage is doomed to fail:

(i) s_1 fails if the execution continues to generate h_2 since no extension of s_1 linearizes h_2;

(ii) s_2 fails if the execution continues to generate h_1 since no extension of s_2 linearizes h_1; and

(iii) s_3 fails if the execution continues to generate h_2 since no extension of s_3, where write operations are only added after the write operation in s_3, linearizes h_1. ⌟

We denote by $\mathcal{I}_{\mathsf{ws}}$ the class of write strongly-linearizable register implementations. By Lemma 4.7 (with R ordering histories using the prefix relation on the restriction to writes), $\mathcal{I}_{\mathsf{ws}}$ is downward-closed w.r.t. simulation, and the notion of a complete implementation is well-defined. Algorithm 1 presents our proposed complete implementation for this class. Its construction is inspired by a specification given by Attiya and Enea [3, §6] for capturing the hyperproperties of a specific snapshot implementation [1]. It is a generalization of DLR from §2, where instead of loading twice, the reader repeatedly loads from X as long as new values are observed, and non-deterministically decides which value to return.

▶ Remark 5.3. One can define a sequence $\{I_k\}_{k=1}^{\infty}$ of implementations, all with atomic write, and read that non-deterministically picks between k-loads (so ATR $= I_1$ and DLR $= I_2$). It can be shown that all of these implementations are write strongly-linearizable, but for every k, I_{k+1} does not strongly observationally refine I_k. The WSR implementation is what one gets "at the limit" of this sequence, and every I_k trivially strongly observationally refines WSR.

Algorithm 1 WSR: A complete implementation for write strongly-linearizable registers.

```
Shared Variables: the current value $X$.
Multi-assignments are executed atomically.
Method read()
    $\mathcal{V} \leftarrow \{X\}$;
    do
        $\langle \mathcal{V}_{\text{prev}}, \mathcal{V} \rangle \leftarrow \langle \mathcal{V}, \mathcal{V} \cup \{X\} \rangle$;
    while $\mathcal{V} \neq \mathcal{V}_{\text{prev}}$;
    $out \leftarrow$ pick $v \in \mathcal{V}$;
    return $out$;

Method write($v$)
    $X \leftarrow v$;
    return;
```

▶ **Theorem 5.4.** WSR *is complete for the class of write strongly-linearizable register implementations.*

As a consequence of Thm. 5.4, we obtain that single-writer ABD strongly observationally refines WSR, and so we can use WSR to argue about the hyperproperties of client programs that use single-writer ABD.

6 Complete Implementation for Decisive Linearizability

In this section we identify a novel linearizability criterion, which we call *decisive linearizability*. Then, we present a complete implementation for the corresponding class of register implementations, which can serve as a hyperproperty-preserving specification for any implementation in the class. Using this implementation, we show that multi-writer ABD is decisively linearizable, and that decisive linearizability (for registers) is weaker than write strong-linearizability.

▶ **Definition 6.1.** Let I be an implementation of an object O and *Spec* be a specification of O. A linearization mapping $L : \mathsf{E}(I) \to \mathit{Spec}$ is *decisive* if $L(e_1) \preceq_{\mathsf{S}} L(e_2)$ whenever $e_1 \preceq_{\mathsf{P}} e_2$. We say that I is *decisively linearizable* w.r.t. *Spec* if there there exists a decisive linearization mapping $L : \mathsf{E}(I) \to \mathit{Spec}$.

Decisive linearizability, like strong and write strong-linearizability, requires the linearization process to be "online". Nevertheless, unlike strong and write strong-linearizability, it does not require that the sequences of linearizations produced in this process are increasing "at the end", thus allowing operations to be added to the linearized history possibly before operations that are already included in the linearized history. The only requirement of decisive linearizability is that this process maintains the relative order of already linearized operations: once the order between o_1 and o_2 has been decided, it cannot be reverted.

▶ **Example 6.2.** All implementations in §2 are decisively linearizable: ATR and DLR are already write strongly-linearizable (which is a stronger condition, as we show below) and for TNSR and ABD, which are not write strongly-linearizable, this will be proven later in the section. To illustrate how a suitable linearization mapping can be obtained for these implementations, we revisit the histories h and its extensions h_1 and h_2 from Example 5.2. To linearize h, we can pick s_3; later on, if the execution continues according to h_1, we append w2 to the linearization, and if the execution continues to h_2, we add w2 to the linearization before w1 – note that decisive linearizability allows this; finally, when the read returns we add it immediately after the corresponding write.

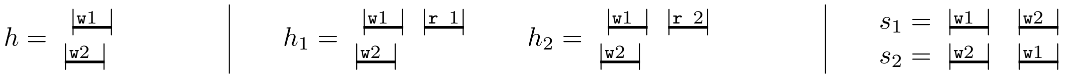
For a "non-example", we use the histories associated with the set T_3 from Fig. 1 to show that the complete implementation for the class of linearizable registers (see Prop. 4.3) is not decisively linearizable. Consider the following history h, its two possible extensions h_1 and

Algorithm 2 DR: A complete implementation for decisively linearizable registers.

Shared Variables: the current value X, the current version number Ver, and a lock flag L.
await B **do** C blocks until the condition B is met, at which point the evaluation of B and the body C are atomically executed. Multi-assignments and **atomic** blocks are executed atomically.

```
Method read()
    await $L = 0$ do $\langle s, \mathcal{V}\rangle \leftarrow \langle Ver, \{X\}\rangle$;
    do
        atomic
            $\mathcal{V}_{\text{prev}} \leftarrow \mathcal{V}$;
            if $Ver \geq s$ then $\mathcal{V} \leftarrow \mathcal{V} \cup \{X\}$;
    while $\mathcal{V} \neq \mathcal{V}_{\text{prev}}$;
    $out \leftarrow$ pick $v \in \mathcal{V}$;
    return $out$;

Method write($v$)
    await $L = 0$ do $s \leftarrow Ver$;
    if * then
        await $L = 0$ do $\langle X, Ver\rangle \leftarrow \langle v, Ver + 1\rangle$;
    else
        await $L = 0 \wedge Ver > s$ do
            $\langle L, tmp, X, Ver\rangle \leftarrow \langle 1, X, v, Ver - 1\rangle$;
        $\langle L, X, Ver\rangle \leftarrow \langle 0, tmp, Ver + 1\rangle$;
    return;
```

h_2, and its possible linearizations s_1 and s_2:



Recall that in the complete implementation for standard linearizability, an execution e that induces h can be extended both to an execution e_1 that induces h_1 and to an execution e_2 that induces h_2. (In particular, this means that an adversary for P_3 from Fig. 1 can decide between these options after the coin toss, refuting the hyperproperty discussed in §2, which is satisfied when each of ATR, DLR, TNSR, ABD and in fact any decisively linearizable implementation is used.) If $L(e) = s_1$ then the linearization of e_1 must reorder the writes in s_1, violating decisiveness. Similarly, if $L(e) = s_2$, then the linearization of e_2 must reorder the writes in s_2, violating decisiveness. Thus, no decisive linearization mapping exists. ⌟

By Lemma 4.7 (with R being the subsequence relation), the class of decisively linearizable implementations is downward-closed w.r.t. simulation and a complete implementation exists for it, for any object. Next, we present a complete implementation for the class of decisively linearizable register implementations. We note that while Definition 6.1 is not specific to registers (unlike Definition 5.1) and Lemma 4.7 applies to any object, the complete implementation we present is only for register implementations. We denote by $\mathcal{I}_{\mathsf{d}}$ the class of all decisively linearizable register implementations. The complete implementation, DR, is presented in Algorithm 2.

DR stores the current value in X and a corresponding version number in Ver. Reads use repeated loads similarly to WSR, but add loaded values to $\mathcal{V}$ only when their version number is not older than the version number when the read started (stored in s). The return value is picked non-deterministically from $\mathcal{V}$.

Writes are based on the idea used in TNSR, allowing stores to non-deterministically choose to be overwritten by a concurrent write, with two important differences. First, new stores by concurrent writes are identified based on version number ($Ver > s$) rather than values (to avoid data dependencies). Second, even if a write chooses to be overwritten, the store to X is not skipped but momentarily executed with a lower version number, to allow concurrent reads to observe it. This is done by a step that temporarily decreases Ver and stores the input value to X, followed by a step that restores Ver and X to their newer values. The two steps are not executed atomically, letting concurrent reads to load the intermediate value. Importantly, a lock L is used to prevent concurrent methods from setting their start version number (s) to a temporary version number, and from updating Ver based on a temporary version number.

```
write(1);       ‖ write(2);       ‖ c ← read();
a ← coin();     ‖ barrier();      ‖ barrier();
barrier();      ‖ b ← read();     ‖
```

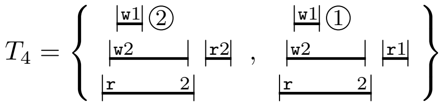


Figure 4 A program P_4 and a set T_4 of traces of the program.

To simplify the presentation, the pseudo-code is written such that a write makes the non-deterministic choice whether to be overwritten or not before it determines that it can indeed be overwritten. As a result, the execution may get stuck. This does not affect linearizability, and this behavior is impossible in our formulation of DR as an LTS.

▶ **Example 6.3.** Allowing concurrent reads to observe “overwritten” writes is crucial for capturing all behaviors of decisively linearizable implementations such as multi-writer ABD. Consider the program P_4 and set of traces T_4 in Fig. 4. The program P_4 extends P_2 from Fig. 1 with another thread, and T_4 is similar to T_2 except that the additional thread observes the value 2 written by the middle thread, even when this value ends up being overwritten. Recall that T_2 can be generated by an adversary for both TNSR and ABD. For TNSR, this leverages the ability of the adversary to postpone the decision whether to store 2 or not until after the coin toss. In contrast, T_4 is not possible for TNSR, since in the trace where the middle thread reads 1, it must be the case that TNSR chose to overwrite 2 and as a result has never stored 2 to X, preventing concurrent threads from loading the value before it is overwritten. (DR does perform a store in such a case, allowing T_4.) Unlike TNSR, ABD allows this behavior: The adversary acts on the left and middle processes similarly to the adversary for P_2 that generates T_2 described in §2, with the added right process sending an additional query when write(2) does so, immediately receiving replies with the initial value from a set of process that excludes the middle process and is one-short from a quorum. Then, when write(2) sends its update, it also replies to the right process with the timestamp it chose. Regardless of the chosen timestamp, it is larger than the initial timestamp, causing the right process to return the value 2. ⌟

▶ **Theorem 6.4.** DR *is complete for the class of decisively linearizable register implementations.*

While not immediate from the definitions, a corollary of Theorems 5.4 and 6.4, together with the observation that WSR $\sqsubseteq_{\mathsf{F}}$ DR, is that every write strongly-linearizable implementation is also decisively linearizable. That is, decisive linearizability is indeed weaker than write strong-linearizability.

Having constructed a complete implementation, we now leverage it to show that other implementations are decisively linearizable: all we need to do is prove that they are simulated by DR. For example, TNSR is trivially simulated by DR, and is therefore decisively linearizable. We show that the same holds for multi-writer ABD.

▶ **Theorem 6.5.** ABD $\sqsubseteq_{\mathsf{F}}$ DR.

▶ **Corollary 6.6.** ABD *is decisively linearizable.*

Thus, DR provides a shared-memory specification for multi-writer ABD that enables reasoning about its hyperproperties.

7 Related and Future Work

Since the observation that linearizability does not suffice for reasoning about randomized client programs and the introduction of strong linearizability [14], many works have studied (im)possibility of implementing strongly linearizable objects under different progress conditions. Helmi et al. [16] showed that lock-free strongly linearizable multi-writer registers, max registers, snapshots, and counters cannot be constructed from a single-writer registers. Attiya et al. [5] and Chan et al. [8] adapted and extended these results for a fault-tolerant message passing setting.

Attiya et al. [6] developed a methodology of making existing implementations probabilistically close to strongly linearizable ones by repeating an effect-free preamble of every method and picking uniformly at random which outcome to continue with. They introduced a correctness condition called *tail strong linearizability* that ensures the effectiveness of this construction. This criterion depends on the choice of the preamble and is thus not comparable to decisive linearizability. Interestingly, the construction in [6] is not effective for our complete implementations (`WSR` and `DR`).

The work of Hadzilacos et al. [15] is closer to our work in its aim to give up strong linearizability, and study what existing implementations do provide. In addition to what we have already discussed, [15, Algorithm 4] demonstrated a multi-writer register implementation that is not write strongly-linearizable. This implementation is essentially a simplified version of `ABD`, and using forward simulation to `ABD`, one can conclude that it is decisively linearizable.

As discussed in length, our work is heavily inspired by [3] that uncovered the correspondence between strong observational refinement and simulation, and suggested the use of non-atomic specifications for reasoning about non-strongly-linearizable implementations. Derrick et al. [10] and Dongol et al. [12] identified a gap in the way [3] handle infinite traces, and show that in that case, while simulation is still necessary for strong observational refinement, only a stronger relation, called *(weak) progressive forward simulation* is sufficient. We focus solely on finite traces, leaving infinite traces to future work.

Bouajjani et al. [7] used forward simulations to non-atomic reference implementations as means to establish linearizability. In particular, they developed abstract stack and queue specifications such that forward simulations to these specifications is necessary and sufficient for establishing linearizability. In our terms, this gets close to complete implementations for the class of linearizable stacks and queues, but, their results are, however, limited to implementations that have explicit marking of linearization points (or so-called "commit points") in some of the methods. Their implementations are highly beneficial in simplifying (and possibly automating) complex linearizability arguments, as the ones needed for Herlihy&Wing Queue [17] and the Time-Stamped Stack [11].

Finally, we note that although we focused on registers, decisive linearizability is a general correctness criterion. Investigating its applicability beyond registers is left for future work. We believe that various implementations that are not strongly linearizable are still decisively linearizable (but, there are known implementations that are not even decisively linearizable, such as the Time-Stamped Stack [11], which allows concurrent complete push operations to remain unordered until a later pop determines their order). Identifying complete implementations for the class of decisively linearizable implementations of other objects is an important (and challenging!) avenue for future work. It would also be interesting to study (im)possibility for decisively linearizable implementations with different progress guarantees.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.

2 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995. doi:10.1145/200836.200869.

3 Hagit Attiya and Constantin Enea. Putting strong linearizability in context: Preserving hyperproperties in programs that use concurrent objects. In *DISC*, volume 146 of *LIPIcs*, pages 2:1–2:17, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2019.2.

4 Hagit Attiya and Constantin Enea. Putting strong linearizability in context: Preserving hyperproperties in programs that use concurrent objects. *CoRR*, abs/1905.12063, 2019. arXiv:1905.12063.

5 Hagit Attiya, Constantin Enea, and Jennifer L. Welch. Impossibility of strongly-linearizable message-passing objects via simulation by single-writer registers. In *DISC*, volume 209 of *LIPIcs*, pages 7:1–7:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.DISC.2021.7.

6 Hagit Attiya, Constantin Enea, and Jennifer L. Welch. Blunting an adversary against randomized concurrent programs with linearizable implementations. In *PODC*, pages 209–219. ACM, 2022. doi:10.1145/3519270.3538446.

7 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving linearizability using forward simulations. In *CAV*, volume 10427 of *LNCS*, pages 542–563. Springer, 2017. doi:10.1007/978-3-319-63390-9_28.

8 David Yu Cheng Chan, Vassos Hadzilacos, Xing Hu, and Sam Toueg. An impossibility result on strong linearizability in message-passing systems. *CoRR*, abs/2108.01651, 2021. arXiv:2108.01651.

9 Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.

10 John Derrick, Simon Doherty, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. Brief announcement: On strong observational refinement and forward simulation. In *DISC*, volume 209 of *LIPIcs*, pages 55:1–55:4. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.DISC.2021.55.

11 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In *POPL*, pages 233–246. ACM, 2015. doi:10.1145/2676726.2676963.

12 Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. Weak progressive forward simulation is necessary and sufficient for strong observational refinement. In *CONCUR*, volume 243 of *LIPIcs*, pages 31:1–31:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CONCUR.2022.31.

13 Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51):4379–4398, 2010. URL: https://www.sciencedirect.com/science/article/pii/S0304397510005001, doi:10.1016/J.TCS.2010.09.021.

14 Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *STOC*, pages 373–382, New York, NY, USA, 2011. ACM. doi:10.1145/1993636.1993687.

15 Vassos Hadzilacos, Xing Hu, and Sam Toueg. On register linearizability and termination. In *PODC*, pages 521–531. ACM, 2021. doi:10.1145/3465084.3467925.

16 Maryam Helmi, Lisa Higham, and Philipp Woelfel. Strongly linearizable implementations: possibilities and impossibilities. In *PODC*, pages 385–394. ACM, 2012. doi:10.1145/2332432.2332508.

17 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

18 Prasad Jayanti, Siddhartha Jayanti, Ugur Yavuz, and Lizzie Hernandez. A universal, sound, and complete forward reasoning technique for machine-verified proofs of linearizability. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi:10.1145/3632924.

19 Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *FTCS*, pages 272–281. IEEE Computer Society, 1997. doi:10.1109/FTCS.1997.614100.

20 Gal Sela, Maurice Herlihy, and Erez Petrank. Brief announcement: Linearizability: A typo. In *PODC*, pages 561–564, New York, NY, USA, 2021. ACM. doi:10.1145/3465084.3467944.

21 Yoav Ben Shimon, Ori Lahav, and Sharon Shoham. Hyperproperty-preserving register specifications (extended version), 2024. doi:10.48550/arXiv.2408.11015.

A Proof Sketches

Proof (sketch) of Lemma 4.7. For downward closure, we observe that $I \sqsubseteq_{\mathsf{F}} I'$ implies that there exists a mapping $\pi : \mathsf{E}(I) \to \mathsf{E}(I')$ such that $\mathsf{h}(e) = \mathsf{h}(\pi(e))$ for every $e \in \mathsf{E}(I)$, and $\pi(e_1) \preceq_{\mathsf{P}} \pi(e_2)$ whenever $e_1 \preceq_{\mathsf{P}} e_2$. Then, given a suitable linearization mapping L for I', the composition $L \circ \pi$ is a suitable linearization mapping for I.

A complete implementation for $\mathcal{I}_R(O, \mathit{Spec})$ is similar to the complete implementation for the class of all linearizable implementations presented in the proof of Prop. 4.3, except that in addition to tracking in its internal state the history h it has observed so far, it also tracks a linearization s of h. When executing an invocation or response α, the linearization is non-deterministically updated to a linearization s' of $h \cdot \alpha$ such that $\langle s, s' \rangle \in R$. If such s' does not exist, the α step is not enabled. This construction generalizes the implementation in [4, Appendix C] which uses the same set of states but only allows to append actions to the linearization. ◀

Proof (sketch) of Theorem 5.4. To show that $\mathtt{WSR} \in \mathcal{I}_{\mathsf{ws}}$, we construct a linearization mapping $L : \mathsf{E}(\mathtt{WSR}) \to \mathsf{Spec}_{\mathsf{Reg}}$ by assigning "linearization points": Write operations that have already stored their values in X are linearized to the transition where they stored this value, and reads that picked a value to return are linearized to the transition where they first loaded that value. Other pending operations are not included in the linearization.

Hardness is much more challenging. Given a write strongly-linearizable implementation I, we begin by instrumenting the state of I with a ghost variable that tracks the full execution performed so far. Then, given an execution e and a transition t, we compare $L(e)$ and $L(e \cdot t)$, where L is the given write strong-linearization mapping for I. A naive attempt to show $I \sqsubseteq_{\mathsf{F}} \mathtt{WSR}$ would execute a store in $\mathtt{WSR}$ at the time the corresponding write operation w is added to the linearization. This fails since w might be added to linearization immediately after all previous writes have been linearized, which can be before w takes effect, and performing the store of $\mathtt{WSR}$ at that step will not allow later reads (that are concurrent with w) to load earlier values.

To overcome this, we prove the existence of a so-called *lazy* linearization mapping. Informally, this mapping adds operations to the linearization only when it must, e.g., when an operation completes, or when a write is needed to justify a completed read. More concretely, assuming arbitrary write strong-linearization mapping L, we prove the existence of a write strong-linearization mapping $L^* : \mathsf{E}(I) \to \mathsf{Spec}_{\mathsf{Reg}}$ with the following additional properties:

1. $L^*(e) = L^*(e \cdot t)$ for every $e \cdot t \in \mathsf{E}(I)$ such that the transition t is not labeled with a response.
2. For every $e \in \mathsf{E}(I)$ and operation o in $L^*(e)$, if o is not completed in $\mathsf{h}(e)$, then it is a write operation and it is not last in $L^*(e)$.
3. L^* is decisive.

Using L^*, the simulation works. Invocation and response transitions are simulated by an identical invocation or response, where invocations of read operations also load the stored value once. The stores in write operations are executed when the write operations are added to the lazy linearization, after which all pending reads that did not already load the stored value load it.

The crux of the proof is to justify that when a completed read is added to the linearization, the matching read in WSR has already loaded the value it needs to pick to match the return value of that read. For this, it suffices to show that the value the read returns was written by a write that either was rightmost in the linearizaton of the prefix of the execution up to the transition that invoked the read, or was added to the linearization later (as these are exactly the writes whose stores we load as described above). The properties of the lazy linearization are used to establish this fact. ◀

Proof (sketch) of Theorem 6.4. For inclusion, DR $\in \mathcal{I}_{\mathsf{d}}$, given an execution of DR we construct a linearization that only includes writes that already stored their value in X and reads that picked a value to return. This is similar to the linearization in the proof that WSR $\in \mathcal{I}_{\mathsf{ws}}$. However, the order in which the operations are linearized is more involved. We begin by assigning to each operation we intend to include in the linearization a version number: for a write operation it is the version number it wrote in the transition where it stored its value into X, and for a read operation it is the version number in the pre-state of the transition where it first loaded the value it later picked to return. Operations are ordered based on version number, with ties broken based on the ordering induced by the aforementioned transitions. The ordering between existing operations according to these rules does not change when new operations are added, and so the mapping we get is decisive. We can use the conditions guarding loads and "roll backs" to earlier version numbers to show the ordering according to the above rules respects real time order.

For hardness, the proof closely follows the proof that WSR is $\mathcal{I}_{\mathsf{ws}}$-hard. The main difference is that when a write appears for the first time in a linearization, it might not appear to the right of writes which already appeared earlier. We use the roll back mechanism to simulate these writes, thus maintaining an invariant that X contains the value of the rightmost write in the linearization. ◀

Proof (sketch) of Theorem 6.5. The simulation keeps track of when a pair of value v and timestamp ts reaches a majority of other processes for the first time. This can happen due to either a write distributing its newly written value or a read distributing its decided read value. When this happens, we check whether ts is larger than the current maximal timestamp that reached a majority of processes. If so, we perform in DR a store of v, attached to a new, larger version number, and then load this value with all threads that are active in a read method. Otherwise, we use the "overwritten value" path of DR: temporarily store v with a lower version number, collect this value by concurrent readers that can see it, and finally restore X to its latest value. ◀

Freeze-Tag in L_1 Has Wake-Up Time Five with Linear Complexity

Nicolas Bonichon
LaBRI, University of Bordeaux, CNRS, Bordeaux INP, France

Arnaud Casteigts
LaBRI, University of Bordeaux, CNRS, Bordeaux INP, France
CS Department, University of Geneva, Switzerland

Cyril Gavoille
LaBRI, University of Bordeaux, CNRS, Bordeaux INP, France

Nicolas Hanusse
LaBRI, University of Bordeaux, CNRS, Bordeaux INP, France

Abstract

The Freeze-Tag Problem, introduced in Arkin et al. (SODA'02) consists of waking up a swarm of n robots, starting from a single active robot. In the basic geometric version, every robot is given coordinates in the plane. As soon as a robot is awakened, it can move towards inactive robots to wake them up. The goal is to minimize the makespan of the last robot, the *makespan*.

Despite significant progress on the computational complexity of this problem and on approximation algorithms, the characterization of exact bounds on the makespan remains one of the main open questions. In this paper, we settle this question for the ℓ_1-norm, showing that a makespan of at most $5r$ can always be achieved, where r is the maximum distance between the initial active robot and any sleeping robot. Moreover, a schedule achieving a makespan of at most $5r$ can be computed in time $O(n)$. Both bounds, the time and the makespan are optimal. Our results also imply for the ℓ_2-norm a new upper bound of $5\sqrt{2}r \approx 7.07r$ on the makespan, improving the best known bound of $(5 + 2\sqrt{2} + \sqrt{5})r \approx 10.06r$.

Along the way, we introduce new linear time wake-up strategies, that apply to any norm and show that an optimal bound on the makespan can always be achieved by a schedule computable in linear time.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases freeze-tag problem, metric, algorithm

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.9

Related Version *Full Version*: http://arxiv.org/abs/2402.03258 [7]

Funding The last three authors are supported by the French ANR, project ANR-22-CE48-0001 (TEMPOGRAL).

Acknowledgements We thank J.S. Mitchell and the anonymous referees for their advice on a previous version of this article.

1 Introduction

In a collaborative swarm of robots, individual robots often have limited capacities in terms of energy, sensing, computation, movement or communication. They cooperate in order to achieve global tasks like exploring a network [11] or planning the motion of each individual robot without conflict [24]. As the robots energy is limited, it may be necessary to switch them off and wake them up later, which requires an efficient way to do so.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 9; pp. 9:1–9:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The Freeze-Tag Problem (FTP) is an optimization problem that consists of activating as fast as possible a swarm of robots represented by points in some metric space (in general, not necessarily Euclidean). Active (or awake) robots can move towards any point of the space at a constant speed, whereas inactive robots are asleep (or frozen) and can be activated only by a robot moving to their position. Initially, there are n sleeping robots and one awake robot. The goal is to determine a schedule whose makespan is minimized; that is, the time until all the robots have been activated is minimized. FTP has applications not only in *robotics*, e.g. with group formation, searching, and recruitment, but also in *network design*, e.g. with broadcast and IP multicast problems. See [4, 3, 18] and references therein.

State of the art. FTP is NP-Hard in high dimension metrics like centroid metrics [3] (based on weighted star n-vertex graphs) or unweighted graph metrics with a robot per node [4]. Many subsequent works have extended this hardness result to constant dimensional metric spaces, including the Euclidian ones. A series of papers [1, 15, 21] proves that FTP is actually NP-Hard in $(\mathbb{R}^3, \ell_p)$, for every $p \geq 1$, i.e., in 3D with any ℓ_p-norm[1]. For 2D spaces, this remains NP-Hard in $L_2 = (\mathbb{R}^2, \ell_2)$, leaving open the question for other norms [1]. It is believed, see [3, Conjecture 28], that FTP remains NP-Hard in $L_1 = (\mathbb{R}^2, \ell_1)$. Beyond being interesting from a theoretical point of view, the L_1 case corresponds to the case where movements are restricted to be orthogonal. This is the case for swarms of robots in certain warehouses.

Several approximation algorithms and heuristics were designed. In their seminal work, [3] developed a 14-approximation for centroid metrics, and a PTAS for $(\mathbb{R}^d, \ell_p)$. The authors of [4] presented a $O(1)$-approximation for unweighted graph metrics with one robot per node, and a greedy strategy analyzed in [25] gives a $O(\log^{1-1/d} n)$-approximation in $(\mathbb{R}^d, \ell_p)$. For general metrics, the best approximation ratio is $O(\sqrt{\log n}\,)$ [18]. For heuristics, several experimental results can be found in [9, 10, 17]. See [5, 20, 14, 8] for generalizations and variants of the problem, including the important online version.

As observed by [2], the FTP can be rephrased as finding a rooted spanning tree on a set of points with minimum depth, where the root node (corresponding to the awake robot) has one child and all the other nodes (corresponding to the n sleeping robots) have at most two children (see Figure 1). Each edge has a non-negative *length*, representing the distance in the metric space between its endpoints. Such a tree is called a *wake-up tree*, and its weighted-depth is called the *makespan*.

This problem can be approached by constructing a bounded-degree B minimum diameter spanning tree (BDST), whose best known approximation is $O(\sqrt{\log_B n})$ [18]. As shown in [3], the BDST problem for $B = 2$ can be approximated within a constant using approximation algorithms for Traveling Salesman Problem, in its *metric* version (hereafter, simply TSP). Moreover, the Path-TSP, a generalization of TSP in which one asks for finding a minimum path length spanning a point set from given start and end points, provides a valid wake-up tree and thus a solution for FTP. The link with TSP is reenforced by the recent approximated reduction of Path-TSP to TSP [26]. In fact, as shown by [18], the BDST problem for $B = 3$, implies the same guarantee for FTP, i.e., $O(\sqrt{\log n})$ times the optimal.

This being said, there are significant differences between TSP and FTP, the latter being considered as a cooperative TSP version where awake robots can help in visiting unvisited cities. First, from an algorithmic point of view, the best lower bound on the approximation

[1] The ℓ_p-norm of a given a vector $u = (u_1, \ldots, u_d) \in \mathbb{R}^d$ is defined by $\ell_p(u) = (\sum_{i=1}^{d} |u_i|^p)^{1/p}$. We denote by $(\mathbb{R}^d, \ell_p)$ the d-dimensional normed linear space where the distance between two points $u, v \in \mathbb{R}^d$ is given by $\ell_p(u - v)$. We denote by L_p the 2D normed linear space $(\mathbb{R}^2, \ell_p)$.

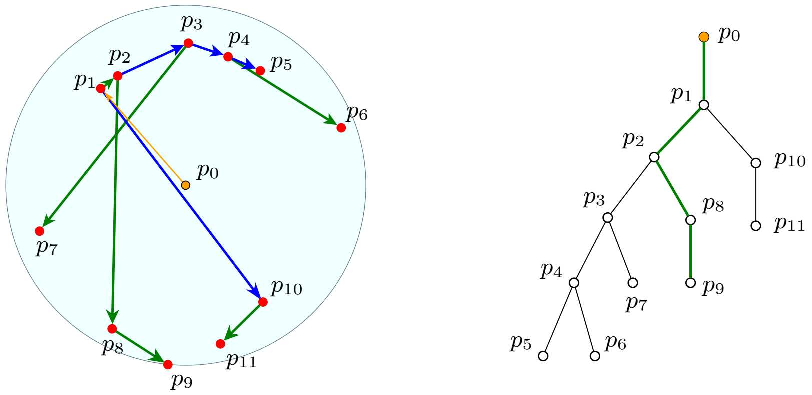


Figure 1 Example of a (here, Euclidean) instance of FTP (on the left). The robot at p_0 must wake up $n = 11$ sleeping robots at $p_1, \ldots, p_n$. In this example, positions are normalized in the unit ℓ_2-disk, p_0 being at the center. An optimal solution, depicted by arrows, can be represented as a binary weighted tree (right). The makespan is the length of the longest (weighted) branch in that tree, here 2.594, corresponding to the path $(p_0, p_1, p_2, p_8, p_9)$. Observe that, even if the sleeping robots are in a convex configuration, the optimal solution may have multiple edge crossings.

factor are $5/3 - \varepsilon$ for FTP [3], and only $123/122 - \varepsilon$ for TSP [16] (assuming $\mathsf{P} \neq \mathsf{NP}$). On the other side, the time complexity for PTAS in $(\mathbb{R}^d, \ell_p)$ is $n(\log n)^{(d/\varepsilon)^{O(d)}}$ for TSP [6, 22, 19] vs. $O(n \log n) + 2^{(d/\varepsilon)^{O(d)}}$ for FTP [3], subject to $\varepsilon \leq \varepsilon_d$, where ε_d depends on the number of dimensions d. Second, and perhaps more fundamentally, it is well known that, even in the unit ball in $(\mathbb{R}^d, \ell_p)$, the shortest spanning path (or tour) has unbounded length in the worst-case (it depends on n), whereas the makespan for FTP is bounded by a constant (that does not depend on n). For TSP, the worst-case length is $\Theta_d(n^{1-1/d})$ [12], whereas for FTP the worst-case optimal makespan is no more than some constant ρ_d, independent of n [3].

The constant ρ_d plays an important role for PTAS and approximation algorithms. For instance, it drives the condition "$\varepsilon \leq \varepsilon_d$" in the grid refinement approach of [3], where local solutions in radius-$(1/\varepsilon)$ balls have to be constructed. For $(\mathbb{R}^2, \ell_2)$, the constant ρ_2 coming from the approach of [3] has been proved to be at most 57 by [27]. The latter authors also construct in time $O(n)$ a wake-up tree of makespan at most $5 + 2\sqrt{2} + \sqrt{5} \approx 10.06$, which is the best known upper bound for ρ_2.

Main contributions. Although some of our results hold for arbitrary norms η, we mostly focus on the ℓ_1-norm assuming that robots are spread in the plane $\mathbb{R}^2$ within a *normalized disk of radius* 1 *centered at its initial awake robot of coordinate* $(0, 0)$. Note that the unit ℓ_2-disk is a usual disk whereas the unit ℓ_1-disk is a square, rotated by 45 degrees with respect to the coordinate axes.

Our results are summarized in Table 1 and deal with lower and upper bounds on the makespan for different norms, as well as their algorithmic solutions. For ℓ_p-norms, the upper bounds mainly come from our new upper bound for ℓ_1 (Theorem 1). Moreover, we show how to build, in linear time, a wake-up tree achieving a makespan of at most the best upper bounds known for arbitrary norm (Theorem 2).

More precisely, we get:

▶ **Theorem 1.** *A robot at the origin can wake up any set of n sleeping robots in the unit ℓ_1-disk with a makespan of at most* 5*. The wake-up tree can be constructed in $O(n)$ time.*

Table 1 Lower and upper bounds on the makespan denoyed by the wake-up constant $\gamma(\eta)$ for different norms η. The number $\pi(\eta) \in [3,4]$ (resp. $\Lambda(\eta) \in [2,\pi(\eta)]$) is the half-perimeter of a unit η-disk (resp. of the largest inscribed parallelogram in the unit η-disk), measured in the η-metric. $\varphi = (1+\sqrt{5})/2$ is the golden ratio. Note that by Theorem 2, all our upper bounds are complemented with a linear time algorithm constructing wake-up trees of makespan no more than these bounds.

Norm	Lower bounds (*Theorem 10*)	Upper bounds	References
ℓ_1,ℓ_∞	5	5	*Theorem 1*, Section 3
ℓ_2		$5+2\sqrt{2}+\sqrt{5} \approx 10.06$	[27]
ℓ_2	$1+2\sqrt{2} \approx 3.83$	$5\sqrt{2} \approx 7.07$	*Corollary 13*, Section 5
ℓ_p	$1+2^{1+\max(1/p,1-1/p)}$	$5 \cdot 2^{\min(1/p,1-1/p)}$	*Corollary 13*, Section 5
Arbitrary η	$1+\Lambda(\eta) \in [3, 1+\pi(\eta)] \subseteq [3,5]$	$3+\varphi\pi(\eta) \leq 9.473$	*Corollary 11*, Section 5

Obviously, if the awake robot is at distance at most r from all the sleeping robots, then by scaling the unit disk with their positions, and by using Theorem 1, one can construct a wake-up tree of makespan of at most $5r$. By a loose argument, this yields also a 5-approximation $O(n)$ time algorithm for $(\mathbb{R}^2, \ell_1)$, since r is a trivial lower bound on the makespan. A similar statement holds for $(\mathbb{R}^2, \ell_\infty)$.

Both bounds in Theorem 1 are optimal: the makespan of 5, and obviously the linear time construction of the wake-up tree. The upper bound of 5 is reached with $n = 4$ sleeping robots at positions $(\pm 1, 0)$ and $(0, \pm 1)$.

By a simple argument, Theorem 1 immediately improves the best known upper bound for the ℓ_2-norm. Indeed (see also Corollary 13), by scaling the unit ℓ_2-disk, we can use the construction of Theorem 1 to obtain a makespan of $5\sqrt{2} \approx 7.07$ for the unit ℓ_2-disk, improving upon the previous 10.06 upper bound of [27].

Our second result concerns algorithmic aspects of the FTP. Theorem 2 (and its simplified version in Corollary 12) states that there is a linear time algorithm that can match the best known upper bound to wake up a unit disk. The result is general enough to hold in any normed linear space $(\mathbb{R}^2, \eta)$, a.k.a. Minkowski plane.

To make the statement of Theorem 2 precise, let us define $\gamma_n(\eta)$ as the worst-case optimal makespan of a wake-up tree for any set of n sleeping robots in the unit η-disk and rooted at the origin. In other words, $\gamma_n(\eta)$ is the best possible upper bound of the makespan to wake up n sleeping robots from an awake robot placed at the origin, in the unit η-disk, all distances being measured according to the η-metric. Finally, let us introduce the *wake-up constant* w.r.t. the η-norm defined by

$$\gamma(\eta) \quad = \quad \sup_{n \in \mathbb{N}} \gamma_n(\eta) \ .$$

Note that the constant ρ_2 introduced above is simply $\gamma(\ell_2)$, the ℓ_2 wake-up constant.

▶ **Theorem 2.** *Let η be any norm and let $\tau > 3$ be any real such that $\tau \geq \gamma(\eta)$. In time $O(n)$, a wake-up tree of makespan at most τ, rooted at the origin, can be built for any set of n points in the unit η-disk.*

So, if one plugs $\eta = \ell_1$ and $\tau = 5$ in Theorem 2, then proving that $\gamma(\ell_1) \leq 5$ becomes sufficient to obtain a linear time construction of a wake-up tree of makespan at most 5 as claimed in Theorem 1. In other words, given Theorem 2, the main Theorem 1 can simply be restated as: $\gamma(\ell_1) \leq 5$. Furthermore, as already explained, the bound of 5 is attained for $n = 4$ sleeping robots, so $\gamma(\ell_1) \geq \gamma_4(\ell_1) = 5$, and the wake-up constant in ℓ_1-norm is thus 5.

To prove Theorem 1 and Theorem 2, we need several intermediate results, which we believe are of independent interest.

Table 2 Makespan and complexity to wake-up, from the origin, n sleeping robots in a cone of arc-length w in the unit η-disk.

Strategy	Makespan	Complexity	References
`Heap-Strategy`	$1 + w \log_2 n$	$O(n)$	*Proposition 7*, Section 4.1
`Split-Cone-Strategy`	$1 + \varphi w$	$O(n \log n)$	*Proposition 8*, Section 5.1
`Linear-Split-Strategy`	$1 + \varphi w + O(w \cdot (\log^3 n)/n)$	$O(n)$	*Proposition 9*, Section 5.1

In particular, we show how to build efficiently wake-up trees with a small makespan in subregions of the unit disk, like truncated cones (that is the part of a cone within the unit disk) of arc length w. The algorithms presented in Table 2 are named with respect to their underlying strategies. The complexity and makespan of `Linear-Split-Strategy` come from a non-trivial combination of two strategies: `Heap-Strategy` and `Split-Cone-Strategy`, introduced in this paper.

Outline. The proof of Theorem 1 is divided into two parts: (1) the upper bound is presented in Section 3 and its proof, based on a strategy called `L1-Strategy`, is constructive; (2) the linear complexity of the construction is then proved in Section 4, based on `Heap-Strategy`. Theorem 2 is proved by the use of similar strategies. In Section 5, we describe `Split-Cone-Strategy` and show how these strategies can be combined to get new bounds for arbitrary norms. Due to space limitations, some of the proofs are deferred to the Full Version [7], while providing a summary of the main ideas in the body of the paper.

2 Preliminaries: from cones to triangles and squares

Most of our algorithms are based on a partitioning of the unit disk into several shapes. The basic shape is a *cone* centered at the initial awake robot $p_0 = O$. Given two points A, B of the unit circle centered at the origin O, we denote by $\mathsf{arc}(A, B)$ the part of the circle that is traversed anti-clockwise from A to B. The length of $\mathsf{arc}(A, B)$ is denoted by $|\mathsf{arc}(A, B)|$, and is called *arc-length*. We have $|\mathsf{arc}(A, B)| \in [0, 2\pi(\eta))$, where $\pi(\eta)$ is the half-perimeter of a unit η-disk. Note that arc-length and half-perimeter are measured in the η-norm. For instance, $\pi(\ell_2) = 3.14...$ and $\pi(\ell_1) = 4$. Then, $\mathsf{cone}(A, w)$, where $w = |\mathsf{arc}(A, B)|$, is the region of the unit η-disk bounded by $\mathsf{arc}(A, B)$ and the segments $[OA]$ and $[OB]$.

Let us focus on L_1. The unit ℓ_1-disk has a four-fold symmetry and has perimeter $2\pi(\ell_1) = 8$. We consider two specific cones: *squares* and *triangles*. See Figure 2. More precisely, and up to scaling and symmetry along the four axis, a *square* is a $\mathsf{cone}(P, 2)$ with a point $P = (1/2, -1/2)$, whereas a triangle is a $\mathsf{cone}(Q, 1)$ where $Q = (1, 0)$. In the unit ℓ_1-disk, each sides of a square has length 1, as well as its diameter (its diagonals). A triangle has also diameter 1 (its hypotenuse), with both sides of length 1, and it forms an isosceles right triangle. Thus, each square region represents a fourth of the unit disk, possibly subdivided further into two equal triangles.

3 The makespan for L_1 is at most 5

At a high level, the proof consists of recruiting first a team of robots in a dense subregion, then these robots can wake up the other regions in parallel. The difficult part is to select these regions (triangles and squares) appropriately, depending on the number of sleeping robots and their distribution, and to prove that the bound holds in all the cases.

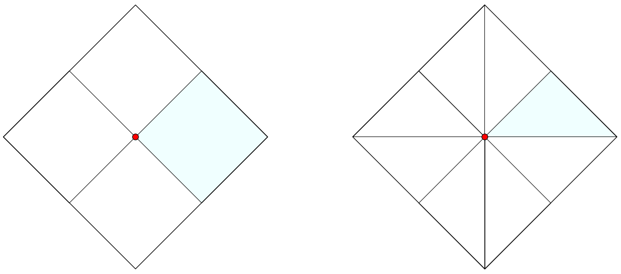


Figure 2 The unit ℓ_1-disk, divided into squares and triangles of diameter 1.

The proof relies on three key lemmas dealing with different wake-up processes in specific subregions, namely:

▶ **Lemma 3.** *A robot located at a corner of a square of diameter one can wake up any number $n \leq 5$ of robots in the square in two time units.*

Unfortunately, Lemma 3 cannot be extended beyond 5 robots. We can show that a makespan of 13/6 is required to wake up some configurations with 6 robots. It seems unavoidable to consider a case-based proof, which motivates the distinction between Lemma 3 and Lemma 4.

▶ **Lemma 4.** *A robot located at a corner of a square of diameter one can wake up 6 robots in the square and return to the origin with these robots in three time units.*

▶ **Lemma 5.** *A robot located at any of the three corners of a triangle T of diameter one, or two robots located at a same point on a side of T (not the hypotenuse) can wake up all the robots in T in two time units.*

A significant part of the paper is devoted to proving these lemmas. In particular, the proof of Lemma 5 is based on a complex recursive algorithm which is divided into 13 subcases.

Equipped with these lemmas, the proof of the main statement can be described as follows.

3.1 Proof of Theorem 1

It is based on the following strategy, called `L1-Strategy`, that is split into four scenarios as follows, depending on the number n_0 of robots in the densest square:

- $\boldsymbol{n_0 = 1}$**.** In this case, there are at most four robots to be awakened. The initiator wakes up one of them in one time unit. We now have two awake robots. Each of them independently wakes up another sleeping robot (if needed), in at most two time units (largest possible distance within the unit ℓ_1-disk). Then, any of the awake robots wakes up the last robot (if any) in at most two time units, which gives a total makespan of at most $1 + 2 + 2 = 5$.
- $\boldsymbol{2 \leq n_0 \leq 5}$**.** We recruit n_0 robots from the densest square S in two time units (Lemma 3), then come back to the origin (by time $2 + 1 = 3$) with $n_0 + 1 \geq 3$ awake robots. Since S is the densest square, three of the awake robots at the origin can each wake up one of the remaining squares (Lemma 3) in two time units, which gives a total of at most $3 + 2 = 5$.
- $\boldsymbol{6 \leq n_0 \leq 10}$**.** We recruit 6 robots (chosen arbitrarily) in the densest square S and move them to the origin in 3 time units (Lemma 4). Together with the initiator, this makes 7 robots. One of them wakes up the remaining robots in S, of which there are at most

4, in two time units (Lemma 3). The 6 others split into three teams of two robots, one team for each remaining square, and each robot wakes up half of the sleeping robots in its assigned square, again in two time units (Lemma 3), which gives a total of at most $3 + 2 = 5$.

- $\boldsymbol{n_0} \geq \mathbf{11.}$ The densest square S must contain a triangle T with at least $\lceil n_0/2 \rceil \geq 6$ sleeping robots. We wake up all the robots of T in 2 time units (Lemma 5) and move them to the origin. This makes at least 7 awaken robots at the origin. Each of them wakes up a remaining triangle in 2 time units (Lemma 5 again), which gives a total of at most $2 + 1 + 2 = 5$ time units.

This completes the upper bound of 5 on the makespan for L_1. Thanks to Theorem 2, a wake-up tree with such a makespan can be constructed in linear time, which completes the proof of Theorem 1.

3.2 Lemmas 3 and 4: monotonic paths

The full proofs of these lemmas are given in the Full Version [7]. We give here a summary of the main ideas. A path $(p_0, p_1, \ldots, p_t)$ is *monotonic* if it is both x-monotonic and y-monotonic. An essential feature of the ℓ_1-norm is that all monotonic paths are shortest paths. In other words, the length of a monotonic path equals the distance between its endpoints. A path is k*-monotonic* if it can be subdivided into at most k consecutive monotonic paths. A key remark is that within a region of diameter δ, the length of a k-monotonic path is at most $k\delta$.

For Lemma 3, we establish that for any set of at most 5 points, there exists a wake-up tree such that every branch is 2-monotonic.

For Lemma 4, a similar approach is taken by incorporating segments that return to the starting point and demonstrating that the resulting paths are 3-monotonic.

Observe that the monotonic nature of a path is solely determined by the relative arrangement of points in terms of their x and y coordinates. Therefore, our considerations are confined to a finite number of configurations (that can be indexed by permutations). See Figure 3.

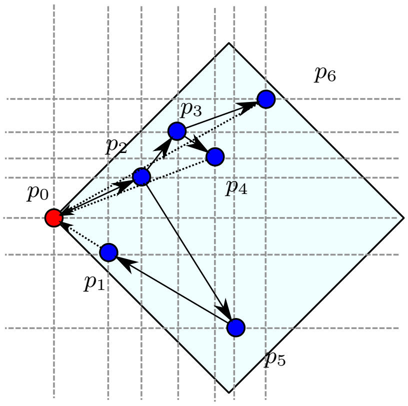


Figure 3 Illustration of a case of Lemma 4, the awake robot being at position p_0. Robots are first ordered w.r.t. their x-coordinate. p_0, p_2, p_3, p_6 is a 1-monotonic path whereas p_0, p_2, p_3, p_4 is a 2-monotonic path. p_0, p_2, p_5, p_1, p_0 is a 3-monotonic path. This wake-up tree is valid for any set of sleeping robots whose orders relative to the axes correspond to the permutation (3246517): the relative y-order of p_0 is 3, the one of p_1 is 2, ..., the one of p_6 is 7.

3.3 Lemma 5: recursive wake-up in triangles

Lemma 5 establishes that an arbitrary number of sleeping robots in a triangle T of diameter 1 can be woken up within two time units. The approach is inductive. Namely, waking up a triangle often reduces to waking up smaller nested triangles (containing strictly less sleeping robots), which explains why the statement of the lemma addresses several starting configurations. We present here a representative subset of three cases (out of thirteen). The other cases are presented in the Full Version [7].

▶ **Lemma 5.** *A robot located at any of the three corners of a triangle T of diameter one, or two robots located at a same point on a side of T (not the hypotenuse) can wake up all the robots in T in two time units.*

Because the unit ℓ_1-disk has a four-fold symmetry, we assume that the triangle T is oriented as in Figure 4, with vertices ABC and hypotenuse $[BC]$.

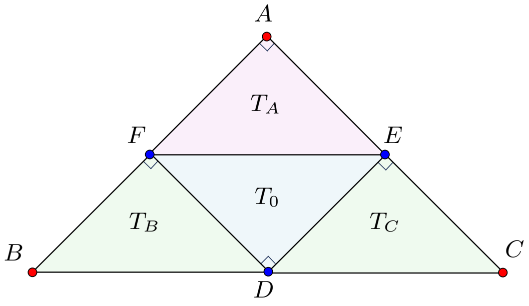


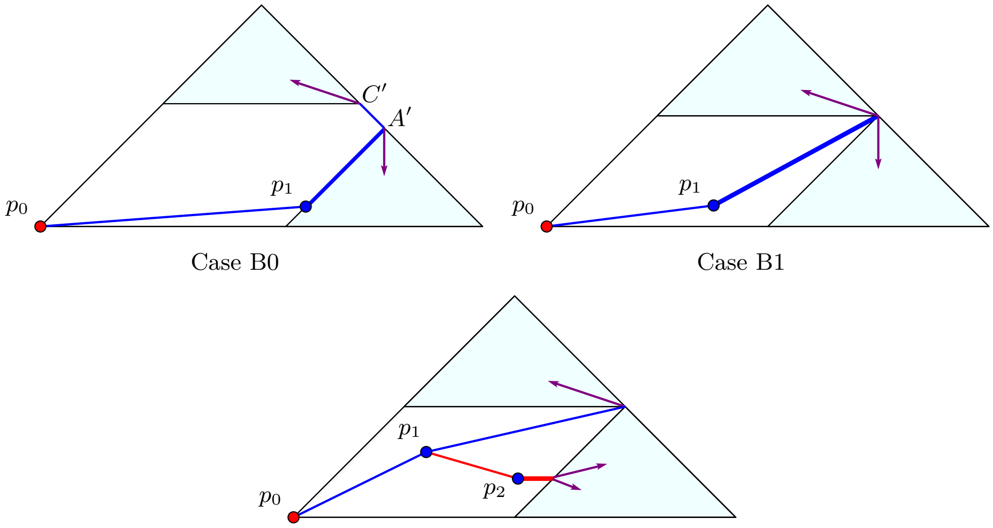
Figure 4 The triangle T with vertices $B = (0, 0)$, $C = (1, 0)$ and $A = (1/2, 1/2)$. Subdivision of the triangle T.

The goal is to show that all sleeping robots in T can be woken up in two time units, for each of the possible starting configurations. Up to symmetry, these configurations are: **Case A**: one awake robot is located in A; **Case B**: one awake robot is located in B; **Case C**: two awake robots are located at a same point along segment $[AB]$.

The strategy depends critically on how the robots are distributed within the triangle, which gives rise to a number of subcases (13 overall). Our proof is fully constructive (i.e., it yields an actual quadratic time algorithm that we have implemented) and we show in the Full Version [7] how to get a linear time algorithm. Technically, we proceed by induction on the number of sleeping robots in T. For cases A and B, at least one sleeping robot, at position p_1 in T, will be awake by the robot at $p_0 \in \{A, B\}$. For Case C, this is not necessarilly the case as an awake robot may simply move to another location without waking up any other robot. However, after one application of Case C, we check that Case C cannot immediately reapply. In other words, after two steps of induction, cases A or B will apply with one less sleeping robot.

The proof of Case B and C, and their subcases, relies on a regular subdivision of T into four smaller triangles of equal size. Call D, E, F the middle points of segments $[BC], [CA]$ and $[AB]$, respectively, and let T_A, T_B, T_C, T_0 be the triangles AFE, BDF, CED, and DEF (see Figure 4). Each of these triangles has diameter 1/2. Let $\varangle P_B$ be the parallelogram $BDEF$. The diameter of $\varangle P_B$ is 1.

We now present three of the thirteen cases (see Figure 5). These three cases are representative of the different arguments used. In these cases, the awake robot starts at point B, also referred to as p_0. The case analysis depends on the number of sleeping robots in $\varangle P_B$. Namely, we apply Case B0 if it is empty, B1 if it contains one robot, and B2 if it contains two robots. A graphical summary of these three subcases is shown in Figure 5.



Case B0

Case B1

Case B2

Figure 5 The subcases B0, B1 and B2 of Case B for Lemma 5. Regions in blue correspond to region where recursion occurs. Outgoing purple arrows indicates that the region will be woken up from the head location. A thick edge indicates that two awake robots follow the same path.

- **Case B0.** ▱P_B is empty. We increase the size of ▱P_B homothetically, keeping one of its corners at B, until a point p_1 is found (see Figure 5 - B0). The new parallelogram intersects with $[AC]$ in two points C' and A', where C' is the closest to A. This forms two smaller triangles which are homothetic to ABC. Because they result from intersecting a parallelogram, these triangles have the same size; namely, they have diameter $d = |AC'| = |A'C|$, which also implies that $|C'A'| = 1 - 2d$.
  The wake-up tree is as follows. The initial robot wakes up p_1. Depending on what side of the parallelogram p_1 lies on, both robots reach C' or A' using a path that is still monotonic from p_0, so they arrive before one time unit. One of them then reaches the other point (C' or A') in time $1-2d$. Finally, each robot wakes up one of the two triangles (separately) in time $2d$, recursing into case A and B (respectively). Overall, the makespan is thus $1 + (1 - 2d) + 2d = 2$.
- **Case B1.** ▱P_B contains one robot. The robot at p_0 wakes up this robot, then both robots move to E before one time unit, since the path (p_0, p_1, E) is monotonic. Finally, one of them wakes up T_A (recursing in Case B) and the other T_C (recursing in Case A). These triangles have half the size of T, thus the makespan is at most $1 + 2 \cdot (1/2) = 2$.
- **Case B2.** ▱P_B contains two robots at p_1 and p_2. W.l.o.g., assume $p_1 \leq_x p_2$. If (p_0, p_1, p_2) is monotonic, the strategy is the same as in Case B1: the robots reach point E in one time unit, then two of them wake up T_A and T_C independently. Otherwise, there exists a point C^* of $[DE]$ such that (p_1, p_2, C^*) is 1-monotonic. In this case, the initial robot wakes up the robot in p_1, then moves to E before one time unit and wakes up T_A in $2 \cdot (1/2) = 1$ time unit. Meanwhile, the robot in p_1 wakes up the robot in p_2 and both move to C^*.

  ▷ Claim 6. The 2-monotonic path (p_0, p_1, p_2, C^*) has length at most one.

  Proof. Let $p_1 = (x, y)$ and $C^* = (x', y')$. By 2-monotonicity, the length of the path is $|p_0p_1| + |p_1C^*| = (x + y) + ((x' - x) + (y - y')) = 2y + x' - y'$. In terms of y-coordinate, the height of T is $1/2$, thus the height of ▱P_B is $1/4$, and $y \leq 1/4$. Moreover, because C^* lies on $[DE]$, we have $y' = x' - 1/2$, so $2y + x' - y' \leq 1/2 + x' - (x' - 1/2) = 1$. ◁

We thus have two robots located at C^* before one time unit. These robots can wake up T_C (of diameter $1/2$) in one time unit, by recursing in Case C.

4 Linear time algorithm

4.1 Heap-Strategy and linear time for L_1

A crude analysis of L1-Strategy shows that its complexity is quadratic. Even if we use a suitable data structure, we believe that the current proof cannot lead to a better than $\Omega(n \log n)$ time algorithm.

To obtain a linear algorithm, we will combine it with an algorithm that runs in linear time and achieves a makespan of $3 + O((\log n)/\sqrt{n})$. Therefore, for small values of n (up to a characterized constant), we use L1-Strategy, and for larger values, we use the latter.

The strategy, called Generic-Strategy, is as follows: in a first phase, we divide the disk into $\sqrt{n}$ cones of arc-length $w = 2\pi(\ell_1)/\sqrt{n} = 8/\sqrt{n}$. Initially, we awake the robots in the densest cone using a strategy called Heap-Strategy. This particular cone contains at least $\sqrt{n}$ robots. In a second phase, we assign one awake robot to each of the remaining cones, in parallel, every "sleeping" cone is awaken using Heap-Strategy again. Therefore, the makespan of the Generic-Strategy is no more than $1 + 2C(w, \sqrt{n}) = 1 + 2C(8/\sqrt{n}, \sqrt{n})$, where $C(w, n)$ is the wake-up time for n points in a cone of arc-length w.

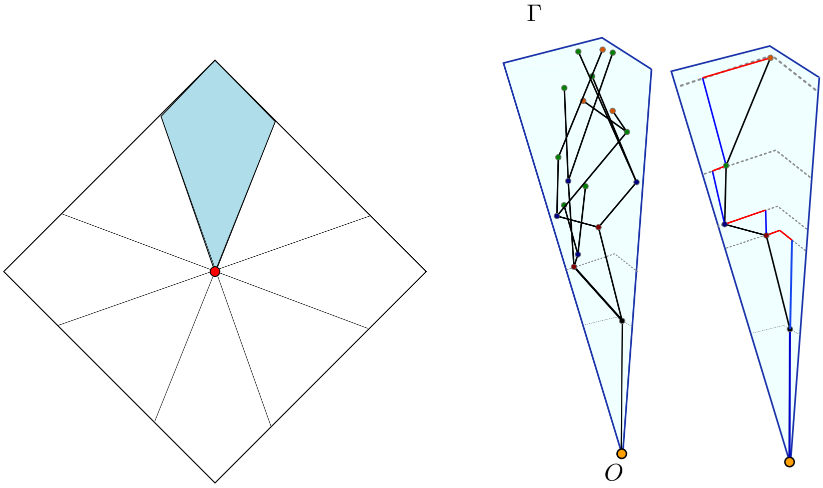


Figure 6 On the left, an arbitrary partition of the unit ℓ_1-disk into cones. In the middle, a wake-up tree computed using Heap-Strategy for an arbitrary cone (Γ represents an arc for an arbitrary norm). On the right, the analysis of the length of a branch drawn in black. In blue (resp. red) the radial (resp. angular) displacement of each edge.

To wake up a cone from p_0 containing points $p_1, p_2, \ldots, p_n$, we use the Heap-Strategy that simply constructs a minimum heap (binary) tree whose key is the ℓ_1-distance from the origin. By design, every path from p_0 to a point p_i has a *non-decreasing distance property* of the nodes w.r.t. the root.

More precisely, Heap-Strategy consists in building a minimum (binary) heap tree H for $\{p_1, \ldots, p_n\}$ where the key of p_i is the distance from p_0 to p_i. The wake-up tree rooted at p_0 is then composed of H itself, plus the edge connecting p_0 to the root of H (its top element), i.e., the closest point from p_0. Using the standard "build-heap" and "heapify" routines, H and thus the wake-up tree can be constructed in time $O(n)$.

Although we focus on L_1 in this section, the following proposition holds for any norm η:

▶ **Proposition 7.** *If P is contained in a cone of arc-length w, then* `Heap-Strategy` *constructs in time $O(n)$ a wake-up tree for P, rooted at the origin, with makespan at most $1 + w \lfloor \log_2 n \rfloor$.*

Proof. We can bound the length of an edge by the radial displacement plus the angular displacement. For each edge, the angular displacement is bounded by the arc-length w (see Figure 6). Furthermore, along a branch, the total radial displacement is bounded by the depth of the cone. Since the wake-up tree is a binary tree, each branch has at most $\lfloor \log_2 n \rfloor$ edges. Hence we obtain a makespan of $1 + w \lfloor \log_2 n \rfloor$ for the `Heap-Strategy` on a cone of arc-length w. ◀

Thus, from Proposition 7, we have $C(w, n) = 1 + w \lfloor \log_2 n \rfloor$, and the makespan of the `Generic-Strategy` for L_1 is smaller than $3 + 16 \log_2 n/\sqrt{n}$. So, for n large enough, that is as soon a $3 + 16 \log_2 n/\sqrt{n} \leq 5$, we can use this generic strategy to get a linear time for L_1, and we use the `L1-Strategy` otherwise, both with a guarantee of 5 on the makespan.

4.2 Linear time for arbitrary norms

We can generalize the generic strategy to arbitrary norms for any admissible bound on the makespan. The main difference is that the threshold $n_0 \leq n$ to use the generic algorithm depends on the wanted upper bound $\tau > 3$ assuming that $\tau \geq \gamma(\eta)$. For L_1, we know that $\gamma(\ell_1) = 5$ and we show a solution for $\tau = 5$.

Thus we get:

▶ **Theorem 2.** *Let η be any norm and let $\tau > 3$ be any real such that $\tau \geq \gamma(\eta)$. In time $O(n)$, a wake-up tree of makespan at most τ, rooted at the origin, can be built for any set of n points in the unit η-disk.*

Proof. Assume that $\tau > 3$ and $\tau \geq \gamma(\eta)$. We use the generic strategy presented for L_1. Since it is well-known that $\pi(\eta) \in [3, 4]$, we compute the least integer n_0 such that[2] $3 + 16 \log_2 n_0/\sqrt{n_0} \leq \tau$. Note that n_0 is a fixed constant, independent of n.

- If $n \geq n_0$, then we can apply `Generic-Strategy` providing a makespan that is less than τ.
- If $n < n_0$, then we use can brute-force algorithm for finding an optimal wake-up tree whose makespan is at most $\gamma(\eta)$ by definition of $\gamma(\eta)$. This is also at most τ by the choice of τ. The number of wake-up trees we have to consider in a brute-force algorithm is at most the number of labeled binary trees on n vertices that is $n! \cdot C_n = (2n)!/(n+1)! \leq (2n_0)!/(n_0+1)! = O(1)$ since n_0 is constant, where $C_n = \binom{2n}{n}/(n+1)$ is the nth Catalan's number. And, checking the makespan of each of these trees costs $O(n) = O(n_0) = O(1)$. ◀

5 Wake-up constants for other norms

Observe that `Heap-Strategy` suffers from having an unbounded makespan of $1 + w \log_2 n$ if $w = \omega(1/\log n)$. We address this as follows. First, we introduce a new strategy: `Split-Cone-Strategy` in order to remove this dependency to get a makespan of[3] $1 + \varphi w$ but

[2] For $\eta = \ell_1$, this integer appears to be $n_0 = 11\,665$. In the detailed proof of Theorem 2, in the Full Version [7], we show that we can use `Linear-Split-Strategy` instead of `Split-Cone-Strategy` in the `Generic-Strategy`, and that it is enough to choose the least n_0 such that $3 + 26/\sqrt{n_0} \leq \tau$. For $\eta = \ell_1$, this improved strategy gives an $n_0 = 169$.

[3] Recall that $\varphi = (1 + \sqrt{5})/2$ is the golden ratio.

running in time $O(n \log n)$. Then, we use Split-Cone-Strategy to get new bounds on the makespan for arbitrary norms. At the end of the section, we give some hints to get a linear time version of Split-Cone-Strategy, the full proof being presented in the Full Version [7].

Notations for arbitrary norms. In this paper, we concentrate our attention to the plane $\mathbb{R}^2$. Given a norm η, the *unit disk w.r.t.* η, or the unit η-disk for short, is the normed linear subspace of $(\mathbb{R}^2, \eta)$ induced by all the points at distance at most one from the origin, where distances are measured according to η, the distance between u and v being $\eta(v - u)$. The unit η-disk can be an arbitrary convex body that is symmetric about the origin. Note that the unit ℓ_2-disk[4] is a usual disk whereas the unit ℓ_1-disk is a square, rotated by 45 degrees with respect to the coordinate axes.

5.1 Split-Cone-Strategy

Let us describe the construction of the wake-up tree corresponding to Split-Cone-Strategy: (1) the initial awake robot located at this origin, wakes up the closest robot located at position p_1 w.r.t. the η-distance; (2) we split the current cone of arc-length w into two subcones of arc-length w/φ and $(1 - 1/\varphi)w$; (3) the current point p is linked to the closest point in the non-decreasing order in each of the two subcones; (4) each subcone is subdivided recursively according to Steps 2 and 3 until every point of the initial cone belongs to the binary wake-up tree.

It turns out that the subcone assigned to a point at depth i in the wake-up tree has an arc-length at most w/φ^i. After a precise analysis of the wake-up tree defined by Split-Cone-Strategy, we get:

▶ **Proposition 8.** *If P is contained in a cone of arc-length w, then* Split-Cone-Strategy *constructs in time $O(n \log n)$ a wake-up tree for P, rooted at the origin, of makespan at most $1 + \varphi w$.*

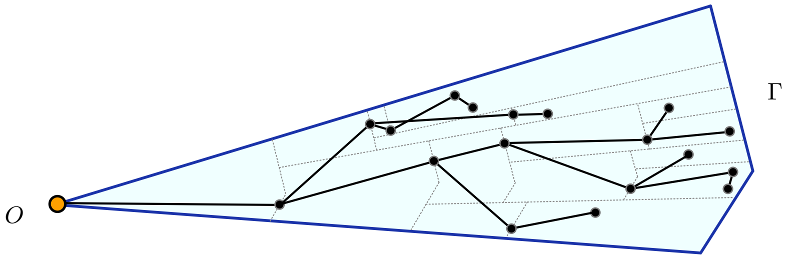


Figure 7 Illustration of the Split-Cone-Strategy, producing non-decreasing wake-up trees. The arc Γ is of length w in the η-norm. Note that edges can cross in such wake-up trees.

Let us sketch the proof of Proposition 8. To analyze the length of the longest branch of the wake-up tree (the makespan), we proceed as in the case of Heap-Strategy: we bound the length of each edge by its radial movement plus its lateral movement. The sum of the radial movements of a branch is at most the radius of the cone, that is 1. In addition, the lateral movement of the i-th edge is at most w/φ^{i-1}. So the sum of the lateral movements along a branch is at most φw. Hence the makespan of Split-Cone-Strategy is at most $1 + \varphi w$.

[4] For convenience, and to avoid extra notation, we use the same “unit η-disk” terminology to denote the normed subspace and, like here, its support, that is the set of all points/vectors of norm at most 1 (the unit disk).

If we combine Split-Cone-Strategy on the $O(n/\log n)$ closest points from the initial position and Heap-Strategy on the remaining points in tiny subcones, we have (see the Full Version [7] for the proofs):

▶ **Proposition 9.** *If P is contained in a cone of arc-length w, the* Linear-Split-Strategy *constructs in time $O(\lambda n)$ a wake-up tree for P, rooted at the origin, with makespan at most $1 + \varphi w + w \cdot (\log_2 n)^3/(\lambda n)$, for every $\lambda \geq 1$.*

5.2 Lower bounds

In L_1, consider four sleeping robots at positions $(\pm 1, 0)$ and $(0, \pm 1)$. Any wake-up tree spanning more than four points must have (unweighted) a depth at least 3. Then, the first hop has length 1, and the next two hops have length 2 (as the four points are mutually at distance 2), which overall gives a makespan of at least 5 for any wake-up tree. In Theorem 10, we give a generalization of this argument for any norm η, leading to an intriguing open question of matching this lower bound for other norms (see Conjecture 14).

Given a norm η, let us define $\Lambda(\eta)$ as half the perimeter of the largest inscribed parallelogram in the unit η-disk measured in the η-metric. (For the ℓ_1-norm, this perimeter corresponds to the circumference of the disk itself, and so $\Lambda(\ell_1) = 4$.) This is a classical parameter of 2D normed spaces. It can be formally defined by (see [23, 13]),

$$\Lambda(\eta) \;=\; \sup_{\substack{u,v \in \mathbb{R}^2 \\ \eta(u),\eta(v) \leq 1}} \{\, \eta(u+v) + \eta(u-v) \,\} \ .$$

It is easy to check that $\Lambda(\eta) \in [2, 4]$. For general norms, the constant $\Lambda(\eta)$ can be difficult to calculate precisely. However, it is known (see [13, Proposition 1] for instance), that, for every $p \in [1, \infty]$, $\Lambda(\ell_p) = 2^{1+\max(1/p, 1-1/p)}$. In particular, $\Lambda(\ell_1) = \Lambda(\ell_\infty) = 4$ and $\Lambda(\ell_2) = 2\sqrt{2}$.

The wake-up constants for $n \in \{0, 1, 2, 3\}$ are easy to calculate. We have $\gamma_0(\eta) = 0$, $\gamma_1(\eta) = 1$, $\gamma_2(\eta) = \gamma_3(\eta) = 3$, and also $\gamma_n(\eta) \geq 3$ for all[5] $n \geq 3$. Our next result gives the exact value for $\gamma_4(\eta)$.

▶ **Theorem 10.** *For any norm η, $\gamma_4(\eta) = 1 + \Lambda(\eta)$.*

This implies a general lower bound of $\gamma(\eta) \geq 1 + \Lambda(\eta)$, for any norm η.

5.3 Upper bounds, ℓ_p-norms, and the conjecture

Wake-up cones of arc-length w with a makespan $1 + \varphi w$ allow us to state a first general upper bound: once two robots are awake at the origin (this can be done in at most two time units), each one can wake up half of the unit disk with arc-length $\pi(\eta)$. By this way, we can bound the wake-up constant for every norm:

▶ **Corollary 11.** *For any norm η, $\gamma(\eta) < 3 + \varphi\pi(\eta) \leq 9.473$.*

Note that since $\Lambda(\eta) \geq 2$, by letting $\tau = 1 + \Lambda(\eta)$, Theorem 2 simplifies and rewrites in the following meta-theorem:

▶ **Corollary 12.** *For any norm η with $\Lambda(\eta) > 2$, one can construct in time $O(n)$ a wake-up tree of makespan at most $\gamma(\eta)$ for any set of n points in the unit η-disk and rooted at the origin.*

[5] For $n \geq 2$, it is enough to place one sleeping robot at $(1, 0)$ and the $n - 1$ others at $(-1, 0)$.

Now, combining Theorem 1, Theorem 10 (with $\eta = \ell_p$), and standard inclusion arguments of unit ℓ_p-disk, we get the following bounds for the ℓ_p wake-up constant, which are better than what one can get by Corollary 11 with $\pi(\ell_p)$:

▶ **Corollary 13.** *For every $p \in [1, \infty]$, $1 + 2^{1+\max(1/p,1-1/p)} \leq \gamma(\ell_p) \leq 5 \cdot 2^{\min(1/p,1-1/p)}$.*

In the light of the lower bound $\gamma(\eta) \geq 1 + \Lambda(\eta)$ implied by Theorem 10, we propose the following natural conjecture.

▶ **Conjecture 14.** *For any norm η, $\gamma(\eta) = 1 + \Lambda(\eta)$.*

According to Theorem 10, which states that the bound $1 + \Lambda(\eta)$ is reached by $n = 4$ robots and doing some experiments, Conjecture 14 can be captured in the aphorism:

Wake up n robots is quicker than wake up four.

Theorem 1 and Corollary 13 prove the conjecture for $\eta \in \{\ell_1, \ell_\infty\}$. For $\eta = \ell_2$, if true, Conjecture 14 combined with Theorem 2 implies that in time $O(n)$ one can construct a wake-up tree of makespan $1 + 2\sqrt{2} \approx 3.82$. Using an analysis of the `Generic-Strategy` combined with the `Split-Cone-Strategy` (see the Full Version [7]), we can show that Conjecture 14 is true for ℓ_2 whenever $n \geq 551$.

6 Conclusion

In this article, we showed that a wake-up tree can be built in linear time with a makespan at most five for robots in L_1. This wake-up constant "five" is optimal: no strategy can guarantee less than five times the radius under the ℓ_1-norm. Our results imply a new upper bound of 7.07 for the ℓ_2-norm, improving upon the existing bound of 10.06, and so, in linear time. Some of our results are general enough to apply to every norm, implying an upper bound of 9.473 for every norm by introducing new algorithmic strategies, namely `Heap-Strategy` and `Split-Cone-Strategy`. We also showed how to get in linear time a wake-up tree of makespan no more than the wake-up constant, for every norm. The construction could be used in another setting since it provides a subcubic geometric graph with small diameter, namely at most $2\gamma(\eta) < 2 \cdot (3 + \varphi\pi(\eta))$ times the radius of the point set (Corollary 11).

Along the way, we conjecture that, for every norm η, the wake-up constant is $1 + \Lambda(\eta)$, where $\Lambda(\eta)$ is half the perimeter of the largest inscribed parallelogram in the unit disk of $(\mathbb{R}^2, \eta)$. According to our results, the conjecture is equivalent to saying that waking up n robots is always faster than waking up four robots. This conjecture is proved for the special cases of ℓ_1 and ℓ_∞ norms. As a first step towards proving the full conjecture, it would be interesting to determine the status of the ℓ_2-norm whose wake-up constant, according to our conjecture, should be $1 + 2\sqrt{2} \approx 3.82$. Among ℓ_p-norms, ℓ_2 is the norm whose current gap between the upper and lower bounds is the largest.

We also showed that the wake-up constant for fixed n asymptotically decreases with n, i.e., $\gamma_n(\ell_2) < \gamma_4(\ell_2)$ for large n (above 500), but we were unable to show that this inequality occurs for small n (say, 10). Surprisingly, some experiments that we conducted (see the Full Version [7]) show that at least one of the two following statements is wrong: (1) the wake-up constant is reached for points that are equally distributed along the boundary of the unit circle; and (2) for every $n \geq 4$, $\gamma_{n+2}(\ell_2) < \gamma_n(\ell_2)$.

In summary, the main open questions are the following:

- Is the wake-up constant of the ℓ_2-norm equal to $1 + 2\sqrt{2}$?
- Is the wake-up constant of the regular-hexagonal-norm[6] equal to 4?
- Is Conjecture 14 true for all ℓ_p-norms? And if so, is it true every norm?
- Does a linear time PTAS exists?
- What is the status of higher dimensions?

References

1 Zachary Abel, Hugo A. Akitaya, and Yu Jingjin. Freeze tag awakening in 2D is NP-hard. In *27th Annual Fall Workshop on Computational Geometry (FWCG)*, November 2017. URL: `https://www.ams.stonybrook.edu/~jsbm/fwcg17/proceedings.html`.

2 Esther M. Arkin, Michael A. Bender, Sándor P. Fekete, Joseph S.B. Mitchell, and Martin Skutella. The freeze-tag problem: How to wake up a swarm of robots. In *13th Symposium on Discrete Algorithms (SODA)*, pages 568–577. ACM-SIAM, 2002. URL: `http://dl.acm.org/citation.cfm?id=545381.545457`.

3 Esther M. Arkin, Michael A. Bender, Sándor P. Fekete, Joseph S.B. Mitchell, and Martin Skutella. The freeze-tag problem: How to wake up a swarm of robots. *Algorithmica*, 46:193–221, 2006. `doi:10.1007/s00453-006-1206-1`.

4 Esther M. Arkin, Michael A. Bender, Dongdong Ge, Simai He, and Joseph S.B. Mitchell. Improved approximation algorithms for the freeze-tag problem. In *15th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 295–303. ACM Press, June 2003. `doi:10.1145/777412.777465`.

5 Amitai Armona, Adi Avidora, and Oded Schwartz. Cooperative tsp. *Theoretical Computer Science*, 411(31-33):2847–2863, June 2010. `doi:10.1016/j.tcs.2010.04.016`.

6 Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5):753–782, September 1998. `doi:10.1145/290179.290180`.

7 Nicolas Bonichon, Arnaud Casteigts, Cyril Gavoille, and Nicolas Hanusse. Freeze-tag in L1 has wake-up time five. Technical Report 2402.03258v1 [cs.DS], arXiv, February 2024. `doi:2402.03258v1`.

8 Josh Brunner and Julian Wellman. An optimal algorithm for online freeze-tag. In *10th International Conference Fun with Algorithms (FUN)*, volume 157 of *LIPIcs*, pages 8:1–11, September 2020. `doi:10.4230/LIPIcs.FUN.2021.8`.

9 Dan George Bucatanschi. The ant colony system for the freeze-tag problem. In *Midstates Conference on Undergraduate Research in Mathematics and Computer Science (MCURCSM)*, pages 61–69, 2004.

10 Dan Georges Bucatanschi, Blaine Hoffmann, Kevin R. Hutson, and R. Matthew Kretchmar. A neighborhood search technique for the freeze tag problem. In *Extending the Horizons: Advances in Computing, Optimization, and Decision Technologies*, volume 37 of *Operations Research/-Computer Science Interfaces Series*, pages 97–113, 2007. `doi:10.1007/978-0-387-48793-9_7`.

11 Shantanu Das. Graph explorations with mobile agents. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities*, volume 11340 of *Lecture Notes in Computer Science*, chapter 16, pages 403–422. Springer, Cham, 2019. `doi:10.1007/978-3-030-11072-7_16`.

12 L. Few. The shortest path and the shortest road through n points. *Mathematika*, 2(2):141–144, 1955. `doi:10.1112/S0025579300000784`.

[6] With this norm, it is easy to show that its unit disk (a regular hexagon) contains inscribed parallelograms of half perimeter 3. This is clearly the largest possible length since 3 is also the half-perimeter of this disk (a hexagon).

13 Jil Gao. Normal structure and the arc length in banach spaces. *Taiwanese Journal of Mathematics*, 5(2):353–366, June 2001. URL: `http://www.jstor.org/stable/43828249`.

14 Mikael Hammar, Bengt J. Nilsson, and Mia Persson. The online freeze-tag problem. In *7th Latin American Symposium on Theoretical Informatics (LATIN)*, volume 3887 of *Lecture Notes in Computer Science*, pages 569–579. Springer, March 2006. `doi:10.1007/11682462_53`.

15 Matthew Johnson. Easier hardness for 3D freeze-tag. In *27th Annual Fall Workshop on Computational Geometry (FWCG)*, November 2017. URL: `https://www.ams.stonybrook.edu/~jsbm/fwcg17/proceedings.html`.

16 Marek Karpinski. Towards better inapproximability bounds for TSP: A challenge of global dependencies. In *Electronic Colloquium on Computational Complexity (ECCC)*, TR15-097, June 2015. URL: `https://eccc.weizmann.ac.il/report/2015/097/`.

17 Hamidreza Keshavarz. Applying tabu search to the freeze-tag. In *1st Conference on Swarm Intelligence and Evolutionary Computation (CSIEC)*, pages 37–41. IEEE Computer Society Press, March 2016. `doi:10.1109/CSIEC.2016.7482136`.

18 Jochen Könemann, Asaf Levin, and Amitabh Sinha. Approximating the degree-bounded minimum diameter spanning tree problem. *Algorithmica*, 41(2):117–129, 2005. `doi:10.1007/s00453-004-1121-2`.

19 Joseph S.B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. *SIAM Journal on Computing*, 28(4):1298–1309, 1999. `doi:10.1137/S0097539796309764`.

20 Zahra Moezkarimi and Alireza Bagheri. A PTAS for geometric 2-FTP. *Information Processing Letters*, 114(12):670–675, 2014. `doi:10.1016/j.ipl.2014.06.017`.

21 Lehilton Lelis Chaves Pedrosa and Lucas de Oliveira Silva. Freeze-tag is NP-hard in 3D with L_1 distance. In *12th Latin-American Algorithms, Graphs and Optimization Symposium (LAGOS)*, volume 223:C, pages 360–366. Procedia Computer Science, September 2023. `doi:10.1016/j.procs.2023.08.248`.

22 Carsten Rössner and Jean-Pierre Seifert. Hardness of approximating shortest integer relations among rational numbers. *Theoretical Computer Science*, 209(1-2):287–297, December 1998. `doi:10.1016/S0304-3975(97)00118-7`.

23 Juan Jorge Schäffer. *Geometry of Spheres in Normed Spaces*, volume 20 of *Lecture Notes in Pure and Applied Mathematics*. Dekker, Marcel, 1976.

24 Trevor Standley. Finding optimal solutions to cooperative pathfinding problems. In *24th AAAI Conference on Artificial Intelligence (AAAI)*, volume 24, pages 173–178. AAAI Press, July 2010. `doi:10.1609/aaai.v24i1.7564`.

25 Marcelo O. Sztainberg, Esther M. Arkin, Michael A. Bender, and Joseph S.B. Mitchell. Theoretical and experimental analysis of heuristics for the "freeze-tag" robot awakening problem. *IEEE Transactions on Robotics*, 20(4):691–701, August 2004. `doi:10.1109/TRO.2004.829439`.

26 Vera Traub, Jens Vygen, and Rico Zenklusen. Reducing path TSP to TSP. In *52nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 14–27. ACM Press, June 2020. `doi:10.1145/3357713.3384256`.

27 Ehsan Najafi Yazdia, Alireza Bagheri, Zahra Moezkarimia, and Hamidreza Keshavarz. An $O(1)$-approximation algorithm for the 2-dimensional geometric freeze-tag problem. *Information Processing Letters*, 115(6-8):618–622, June 2015. `doi:10.1016/j.ipl.2015.02.011`.

Vertical Atomic Broadcast and Passive Replication

Manuel Bravo
Informal Systems, Madrid, Spain

Gregory Chockler
University of Surrey, Guildford, UK

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Alejandro Naser-Pastoriza
IMDEA Software Institute, Madrid, Spain
Universidad Politécnica de Madrid, Spain

Christian Roldán
IMDEA Software Institute, Madrid, Spain

Abstract

Atomic broadcast is a reliable communication abstraction ensuring that all processes deliver the same set of messages in a common global order. It is a fundamental building block for implementing fault-tolerant services using either active (aka state-machine) or passive (aka primary-backup) replication. We consider the problem of implementing reconfigurable atomic broadcast, which further allows users to dynamically alter the set of participating processes, e.g., in response to failures or changes in the load. We give a complete safety and liveness specification of this communication abstraction and propose a new protocol implementing it, called Vertical Atomic Broadcast, which uses an auxiliary service to facilitate reconfiguration. In contrast to prior proposals, our protocol significantly reduces system downtime when reconfiguring from a functional configuration by allowing it to continue processing messages while agreement on the next configuration is in progress. Furthermore, we show that this advantage can be maintained even when our protocol is modified to support a stronger variant of atomic broadcast required for passive replication.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Reconfiguration, consensus, replication

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.10

Related Version *Extended Version*: `https://arxiv.org/abs/2408.08702` [4]

Funding This work was partially supported by the projects BYZANTIUM, DECO and PRODIGY funded by MCIN/AEI, and REDONDA funded by the CHIST-ERA network.

1 Introduction

Replication is a widely used technique for ensuring fault tolerance of distributed services. Two common replication approaches are *active* (aka state-machine) replication [33] and *passive* (aka primary-backup) replication [6]. In active replication, a service is defined by a deterministic state machine and is executed on several replicas, each maintaining a copy of the machine. The replicas are kept in sync using *atomic broadcast* [10], which ensures that client commands are delivered in the same order to all replicas; this can be implemented using, e.g., Multi-Paxos [21].

In contrast, in passive replication commands are executed by a single replica (the *leader* or *primary*), which propagates the state updates induced by the commands to the other replicas (*followers* or *backups*). This approach allows replicating services with non-deterministic operations, e.g., those depending on timeouts or interrupts. But as shown in [3, 17, 19],

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 10; pp. 10:1–10:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

implementing it requires propagating updates from the leader to the followers using a stronger primitive than the classical atomic broadcast. This is because in passive replication, a state update is incremental with respect to the state it was generated in. Hence, to ensure consistency between replicas, each update must be applied by a follower to the same state in which it was generated by the leader. Junqueira et al. formalized the corresponding guarantees by the notion of *primary-order atomic broadcast (POabcast)* [18, 19], which can be implemented by protocols such as Zab [18], viewstamped replication [30] or Raft [31].

The above implementations of atomic or primary-order atomic broadcast require replicating data among $2f + 1$ replicas to tolerate f failures. This is expensive: in principle, storing the data at $f + 1$ replicas is enough for it survive f failures. Since with only $f + 1$ replicas even a single replica failure will block the system, to recover we need to *reconfigure* it, i.e., change its membership to replace failed replicas with fresh ones. Unfortunately, processes concurrently deciding to reconfigure the system need to be able to agree on the next configuration; this reduces to solving consensus, which again requires $2f + 1$ replicas [22]. The way out of this conundrum is to use a separate *configuration service* with $2f + 1$ replicas to perform consensus on configurations. In this way we use $2f + 1$ replicas to only store configuration metadata and $f + 1$ replicas to store the actual data. This *vertical approach*, layering replication on top of a configuration service, was originally proposed in RAMBO [27] for atomic registers and in *Vertical Paxos* [23] for single-shot consensus. Since then it has been used by many practical storage systems [2, 8, 11, 14]. These often use reconfiguration not only to deal with failures, but also to make changes to a functional configuration: e.g., to move replicas from highly loaded machines to lightly loaded ones, or to change the number of machines replicating the service [26, 29, 39].

Unfortunately, while the space of atomic broadcast protocols with $2f + 1$ replicas has been extensively explored, the design of such protocols in vertical settings is poorly understood. Even though one can obtain a vertical solution for atomic broadcast by reducing it to Vertical Paxos, this would make it hard to ensure the additional properties required for passive replication. Furthermore, both Vertical Paxos and similar protocols [3] stop the system as the very first step of reconfiguration, which increases the downtime when reconfiguring from a functional configuration. Due to the absence of a theoretically grounded and efficient atomic broadcast protocol for vertical settings, the designs used in industry are often ad hoc and buggy. For example, until recently the vertical-style protocol used in Kafka, a widely used streaming platform, contained a number of bugs in its failure handling [14]. In this paper we make several contributions to improve this situation.

First, we give a complete safety and liveness specification of *reconfigurable atomic broadcast*, sufficient for active replication (§3). We then propose its implementation in a vertical system with $f + 1$ replicas and an external configuration service, which we call *Vertical Atomic Broadcast (VAB)* (§4). In contrast to prior vertical protocols [3, 23], our implementation allows the latest functional configuration to continue processing messages while agreement on the next configuration is in progress. This reduces the downtime when reconfiguring from a functional configuration from 4 message delays in the prior solutions to 0. We rigorously prove that the protocol correctly implements the reconfigurable atomic broadcast specification, including both safety and liveness.

We next consider the case of passive replication (which we review in §5). We propose *speculative primary-order atomic broadcast (SPOabcast)*, which we show to be sufficient for implementing passive replication in a reconfigurable system (§6). A key novel aspect of SPOabcast is that SPOabcast is able to completely eliminate the downtime induced by a *Primary Integrity* property of the existing POabcast [18, 19]. This property requires the leader of a new configuration to suspend normal operation until an agreement is reached on which messages broadcast in the previous configurations should survive in the new one: in passive replication, these messages determine the initial service state at the leader. Instead, SPOabcast allows the leader to *speculatively* deliver a tentative set of past messages before the agreement on them has been reached, and then to immediately resume normal broadcasting. SPOabcast guarantees that, if a process delivers a message m_2 broadcast by the new leader, then prior to this the process will also deliver every message m_1 the leader speculatively delivered before broadcasting m_2. This helps ensure that the process applies the update in m_2 to the same state in which the leader generated it, as required for the correctness of passive replication.

We show that SPOabcast can be implemented by modifying our Vertical Atomic Broadcast protocol. The use of speculative delivery allows the resulting protocol to preserve VAB's downtime of 0 when reconfiguring from a functional configuration. It thus allows using Vertical Atomic Broadcast to replicate services with non-deterministic operations.

Overall, we believe that our specifications, protocols and correctness proofs provide insights into the principles underlying existing reconfigurable systems, and can serve as a blueprint for building future ones.

2 System Model

We consider an asynchronous message-passing system consisting of an (infinite) universe of processes $\mathcal{P}$ which may fail by *crashing*, i.e., permanently stopping execution. A process is *correct* if it never crashes, and *faulty* otherwise. Processes are connected by reliable FIFO channels: messages are delivered in FIFO order, and messages between non-faulty processes are guaranteed to be eventually delivered. The system moves through a sequence of *configurations*. A configuration C is a triple $\langle e, M, p_i \rangle$ that consists of an epoch $e \in \mathbb{N}$ identifying the configuration, a finite set of processes $M \subseteq \mathcal{P}$ that belong to the configuration, and a distinguished *leader* process $p_i \in M$. We denote the set of configurations by Config. In contrast to static systems, we do not impose a fixed global bound on the number of faulty processes, but formulate our availability assumptions relative to specific configurations (§3).

Reconfiguration is the process of changing the system configuration. We assume that configurations are stored in an external *configuration service (CS)*, which is reliable and wait-free. The configuration service provides three atomic operations. An operation $\texttt{compare\_and\_swap}(e, \langle e', M, p_l \rangle)$ succeeds iff the epoch of the last stored configuration is e; in this case it stores the provided configuration with a higher epoch $e' > e$. Operations $\texttt{get\_last\_epoch}()$ and $\texttt{get\_members}(e)$ respectively return the last epoch and the members associated with a given epoch e.

In practice, a configuration service can be implemented under partial synchrony using Paxos-like replication over $2f + 1$ processes out of which at most f can fail [22] (as is done in systems such as Zookeeper [17]). Our protocols use the service as a black box, and as a result, do not require any further environment assumptions about timeliness [12] or failure detection [7].

3 Specification

In this section we introduce *reconfigurable atomic broadcast*, a variant of atomic broadcast [10] that allows reconfiguration. The broadcast service allows a process to send an *application message* m from a set Msg using a call $\texttt{broadcast}(m)$. Messages are delivered using a notification $\texttt{deliver}(m)$. Any process may initiate system reconfiguration using a call $\texttt{reconfigure}()$. If successful, this returns the new configuration C arising as a result; otherwise it returns $\bot$. Each process participating in the new configuration then gets a notification $\texttt{conf\_changed}(C)$, informing it about C. In practice, `reconfigure` would take as a parameter a description of the desired reconfiguration. For simplicity we abstract from this in our specification, which states broadcast correctness for any results of reconfigurations.

We record the interactions between the broadcast and its users via *histories* h – sequences of *actions* a of one of the following forms:

$$\begin{aligned}&\texttt{broadcast}_i(m), \quad \texttt{deliver}_i(m), \quad \texttt{conf\_changed}_i(C),\\&\texttt{reconfig\_req}_i, \quad \texttt{reconfig\_resp}_i(C), \quad \texttt{introduction}_i(C),\end{aligned}$$

where $p_i \in \mathcal{P}$, $m \in \mathsf{Msg}$ and $C \in \mathsf{Config}$. Each action is parameterized by a process p_i where it occurs (omitted when irrelevant). The first three actions respectively record invocations of `broadcast`, `deliver` and `conf_changed`. The next pair of actions record calls to and returns from the `reconfigure` function. Finally, the `introduction` action records the moment when this function stores the new configuration in the configuration service.

For a history h we let h_k be the k-th action in h, and we write $a \in h$ if a occurs in h. We also write _ for an irrelevant value. We only consider histories where calls to and returns from `reconfigure` match, and a process may perform at most one `introduction` action during the execution of `reconfigure`. For simplicity we assume that all application messages broadcast in a single execution are unique:

$$\forall m, k, l.\ h_k = \texttt{broadcast}(m) \ \land\ h_l = \texttt{broadcast}(m) \implies k = l.$$

For a history h, a partial function $\mathrm{epochOf} : \mathbb{N} \rightharpoonup \mathbb{N}$ returns the epoch of the action in h with a given index. This is the epoch of the latest preceding `conf_changed` at the same process:

$$\begin{aligned}\mathrm{epochOf}(k) = e \iff (\exists i, l, a.\, h_k = a_i \ \land\ h_l = \texttt{conf\_changed}_i(\langle e, \_, \_\rangle) \ \land\ l < k \ \land\\ \forall l'.\ l < l' < k \implies h_{l'} \neq \texttt{conf\_changed}_i(\langle \_, \_, \_\rangle)).\end{aligned}$$

When $\mathrm{epochOf}(k) = e$, we say that the action h_k occurs in e.

Reconfigurable atomic broadcast is defined by the properties over histories h listed in Figure 1. Properties 1 and 2 are self-explanatory. Property 3 ensures that processes cannot deliver messages in contradictory orders. Property 4 disallows executions where sequences of messages delivered at different processes diverge.

The liveness requirements of reconfigurable atomic broadcast are given by Property 5. Property 5a asserts a termination guarantee for reconfiguration requests. As shown by Spiegelman and Keidar [36], wait-free termination is impossible to support even for reconfigurable read/write registers, which are weaker than atomic broadcast. Hence, the guarantee given by Property 5a is similar to obstruction-freedom [15]. Let us say that a configuration C is *activated* when all its members get $\texttt{conf\_changed}(C)$ notifications. Property 5a asserts that, in a run with finitely many reconfigurations, the last reconfiguration request invoked by a correct process and executing in isolation must eventually succeed to introduce a configuration C, which must then become activated if all its members are correct. Properties 5b-c state liveness guarantees for the configuration C similar to those of the classical atomic broadcast. Property 5b asserts that any message broadcast by a (correct) member of C eventually gets delivered to all members of C. Property 5c additionally ensures that the members of C eventually deliver all messages delivered by any process in any configuration.

As in prior work [1, 3, 37], the liveness of our protocols is premised on the following assumption, which limits the power of the environment to crash configuration members.

1. **Basic Configuration Change Properties.**
   a. Any epoch e is associated with unique membership and leader:

   $$\forall e, i, j, M_1, M_2.\ \texttt{conf\_changed}(\langle e, M_1, p_i\rangle) \in h \ \wedge\ \texttt{conf\_changed}(\langle e, M_2, p_j\rangle) \in h \implies p_i = p_j \ \wedge\ M_1 = M_2$$

   b. If a process p_i joins a configuration $C = \langle \_, M, \_\rangle$, then p_i is a member of M:

   $$\forall i, M.\ \texttt{conf\_changed}_i(\_, M, \_) \in h \implies p_i \in M$$

   c. Processes join configurations with monotonically increasing epochs:

   $$\forall e_1, e_2, i, k, l.\ h_k = \texttt{conf\_changed}_i(e_1, \_, \_) \ \wedge\ h_l = \texttt{conf\_changed}_i(e_2, \_, \_) \ \wedge\ k < l \implies e_1 < e_2$$

   d. Any configuration a process joins is introduced; a configuration is introduced at most once:

   $$\forall C.\ (\texttt{conf\_changed}(C) \in h \implies \texttt{introduction}(C) \in h) \ \wedge\ (\forall k, l.\ h_k = \texttt{introduction}(C) \ \wedge\ h_l = \texttt{introduction}(C) \implies k = l)$$

2. **Integrity.** A process delivers a given application message m at most once, and only if m was previously broadcast:

   $$\forall m, i, k, l.\ h_k = \texttt{deliver}_i(m) \ \wedge\ h_l = \texttt{deliver}_i(m) \implies k = l \ \wedge\ \exists j.\ h_j = \texttt{broadcast}(m) \ \wedge\ j < k$$

3. **Total Order.** If some process delivers m_1 before m_2, then any process that delivers m_2 must also deliver m_1 before this:

   $$\forall m_1, m_2, i, j, k, l, l'.\ h_k = \texttt{deliver}_i(m_1) \ \wedge\ h_l = \texttt{deliver}_i(m_2) \ \wedge\ k < l \ \wedge\ h_{l'} = \texttt{deliver}_j(m_2) \implies \exists k'.\ h_{k'} = \texttt{deliver}_j(m_1) \ \wedge\ k' < l'$$

4. **Agreement.** If p_i delivers m_1 and p_j delivers m_2, then either p_i delivers m_2 or p_j delivers m_1:

   $$\forall m_1, m_2, i, j.\ \texttt{deliver}_i(m_1) \in h \ \wedge\ \texttt{deliver}_j(m_2) \in h \implies (\texttt{deliver}_i(m_2) \in h \ \vee\ \texttt{deliver}_j(m_1) \in h)$$

5. **Liveness.** Consider an execution with finitely many reconfiguration requests (`reconfig_req`), and let r be the last reconfiguration request to be invoked. Suppose that r is invoked by a correct process and no other reconfiguration call takes steps after r is invoked. Then r terminates, having introduced a configuration $C = \langle e, M, p_i\rangle$: $\texttt{reconfig\_resp}(C)$. Furthermore, if all processes in M are correct, then:
   a. all processes in M deliver $\texttt{conf\_changed}(C)$;
   b. if $p_i \in M$ broadcasts m while in e, then all processes in M eventually deliver m;
   c. if a process delivers m, then all processes in M eventually deliver m.

Figure 1 Properties of reconfigurable atomic broadcast over a history h.

▶ **Assumption 1** (Availability). *Let $C = \langle e, M, \_\rangle$ be an introduced configuration, i.e., such that $\texttt{introduction}(C) \in h$. Then at least one member of M does not crash before another configuration $C' = \langle e', \_, \_\rangle$ with $e' > e$ is activated.*

Our protocols use the period of time when some member of M is guaranteed not to crash to copy its state to the members of a new configuration.

Finally, we note that in the case of a single static configuration, our specification in Figure 1 corresponds to the classical notion of atomic broadcast [10].

```
1  epoch $\leftarrow 0 \in \mathbb{Z}$
2  new_epoch $\leftarrow 0 \in \mathbb{Z}$
3  next $\leftarrow 0 \in \mathbb{Z}$
4  init_len $\leftarrow -1 \in \mathbb{Z}$
5  last_delivered $\leftarrow -1 \in \mathbb{Z}$
6  members $\in 2^{\mathcal{P}}$
7  leader $\in \mathcal{P}$
8  msg[] $\in \mathbb{N} \to$ Msg $\cup \{\bot\}$
9  status $\in \{$LEADER, FOLLOWER, FRESH$\}$

10 function broadcast($m$):
11     send FORWARD($m$) to leader

12 when received FORWARD($m$) from $p_j$
13 // function broadcast($m$):
14     pre: $p_i$ = leader
15     msg[next] $\leftarrow m$
16     send ACCEPT(epoch, next, $m$)
         to members $\setminus \{p_i\}$
17     next $\leftarrow$ next + 1

18 when received ACCEPT($e, k, m$) from $p_j$
19     pre: status = FOLLOWER $\wedge$ epoch = $e$
20     msg[$k$] $\leftarrow m$
21     send ACCEPT_ACK($e, k$) to $p_j$

22 when received ACCEPT_ACK($e, k$)
     from all members $\setminus \{p_i\}$
23     pre: status = LEADER $\wedge$ epoch = $e$
24     send COMMIT($e, k$) to members

25 when received COMMIT($e, k$)
26     pre: status $\in \{$LEADER, FOLLOWER$\}$ $\wedge$
             epoch = $e \wedge k$ = last_delivered + 1
27     last_delivered $\leftarrow k$
28     deliver(msg[$k$])
```

Figure 2 Vertical Atomic Broadcast at a process p_i: normal operation.

4 The Vertical Atomic Broadcast Protocol

In Figures 2 and 3 we present a protocol implementing the specification of §3, which we call *Vertical Atomic Broadcast (VAB)* by analogy with Vertical Paxos [23]. For now the reader should ignore the code in blue. At any given time, a process executing the protocol participates in a single configuration, whose epoch is stored in a variable epoch. The membership of the configuration is stored in a variable members. Every member of a given configuration is either the leader or a *follower*. A status variable at a process records whether it is a LEADER, a FOLLOWER, or is in a special FRESH state used for new processes. A leader variable stores the leader of the current configuration. We assume that the system starts in an initial active configuration with epoch 0.

Normal operation. When a process receives a call broadcast(m), it forwards m to the leader of its current configuration (line 10). Upon receiving m (line 12), the leader adds it to an array msg; a next variable points to the first free slot in the array (initially 0). The leader then sends m to the followers in an ACCEPT(e, k, m) message, which carries the leader's epoch e, the position k of m in the msg array, and the message m itself.

A process acts on the ACCEPT message (line 19) only if it participates in the corresponding epoch. It stores m in its local copy of the msg array and sends an ACCEPT_ACK(e, k) message to the leader of e. The application message at position k is *committed* if the leader of e receives ACCEPT_ACK messages for epoch e and position k from all followers of its configuration (line 22). In this case the leader notifies all the members of its configuration that the application message can be safely delivered via a COMMIT message. A process delivers application messages in the order in which they appear in its msg array, with last_delivered storing the last delivered position (line 25).

Reconfiguration: probing. Any process can initiate a reconfiguration, e.g., to add new processes or to replace failed ones. Reconfiguration aims to preserve the following invariant, key to proving the protocol correctness.

```
29 function reconfigure():
30     var $e, M, e_{\mathrm{new}}, M_{\mathrm{new}}$
31     $e \leftarrow$ get_last_epoch() at CS
32     $e_{\mathrm{new}} \leftarrow e + 1$
33     repeat
34         if $e \geq 0$ then
35             $M \leftarrow$ get_members($e$) at CS
36             send PROBE($e_{\mathrm{new}}, e$) to $M$
37             wait until received
                   PROBE_ACK(_, $e_{\mathrm{new}}$)
                   from a process in $M$
38             $e \leftarrow e - 1$
39     until received PROBE_ACK(TRUE, $e_{\mathrm{new}}$)
        from some $p_j$
40     $M_{\mathrm{new}} \leftarrow$ compute_membership()
41     if compare_and_swap($e_{\mathrm{new}}-1, \langle e_{\mathrm{new}}, M_{\mathrm{new}}, p_j \rangle$)
        at CS /* introduction($\langle e_{\mathrm{new}}, M_{\mathrm{new}}, p_j \rangle$) */
        then
42         send NEW_CONFIG($e_{\mathrm{new}}, M_{\mathrm{new}}$) to $p_j$
43         return $\langle e_{\mathrm{new}}, M_{\mathrm{new}}, p_j \rangle$
44     else
45         return $\bot$

46 when received PROBE($e_{\mathrm{new}}, e$) from $p_j$
47     pre: $e_{\mathrm{new}} \geq$ new_epoch
48     new_epoch $\leftarrow e_{\mathrm{new}}$
49     if epoch $\geq e$ then
50         send PROBE_ACK(TRUE, $e_{\mathrm{new}}$) to $p_j$
51     else
52         send PROBE_ACK(FALSE, $e_{\mathrm{new}}$) to $p_j$
```

```
53 when received NEW_CONFIG($e, M$)
    from $p_j$
54     pre: new_epoch $= e$
55     status $\leftarrow$ LEADER
56     epoch $\leftarrow e$
57     members $\leftarrow M$
58     leader $\leftarrow p_i$
59     next $\leftarrow \max\{k \mid$ msg$[k] \neq \bot\} + 1$
60     init_len $\leftarrow$ next $- 1$
61     conf_changed($e, M, p_i$)
62     // conf_changed($e, M, p_i$,
           msg[last_delivered+1..init_len])
63     send NEW_STATE($e$, msg, $M$)
        to members $\setminus \{p_i\}$

64 when received NEW_STATE($e, msg, M$)
    from $p_j$
65     pre: new_epoch $\leq e$
66     status $\leftarrow$ FOLLOWER
67     epoch $\leftarrow e$
68     new_epoch $\leftarrow e$
69     msg $\leftarrow msg$
70     leader $\leftarrow p_j$
71     conf_changed($e, M, p_j$)
72     // conf_changed($e, M, p_j, \bot$)
73     send NEW_STATE_ACK($e$) to $p_j$

74 when received NEW_STATE_ACK($e$)
    from all members $\setminus \{p_i\}$
75     pre: new_epoch $=$ epoch $= e$
76     for $k = 1$..init_len do
        send COMMIT($e, k$) to members
```

Figure 3 Vertical Atomic Broadcast at a process p_i: reconfiguration.

▶ **Invariant 1.** *Assume that the leader of an epoch e sends* COMMIT(e, k) *while having* msg$[k] = m$. *Whenever any process p_i has* epoch $= e' > e$, *it also has* msg$[k] = m$.

The invariant ensures that any application message committed in an epoch e will persist at the same position in all future epochs e'. This is used to establish that the protocol delivers application messages in the same order at all processes.

To ensure Invariant 1, a process performing a reconfiguration first *probes* the previous configurations to find a process whose state contains all messages that could have been committed in previous epochs, which will serve as the new leader. The new leader then transfers its state to the followers of the new configuration. We say that a process is *initialized* at an epoch e when it completes the state transfer from the leader of e; it is at this moment that the process assigns e to its epoch variable, used to guard the transitions at lines 18, 22, 25. Our protocol guarantees that a configuration with epoch e can become activated only after all its members have been initialized at e. Probing is complicated by the fact that there may be a series of failed reconfiguration attempts, where the new leader fails before initializing all its followers. For this reason, probing may require traversing epochs from the current one down, skipping epochs that have not been activated.

In more detail, a process p_r initiates a reconfiguration by calling `reconfigure` (line 29). The process picks an epoch number e_{new} higher than the current epoch stored in the configuration service and then starts the probing phase. The process p_r keeps track of the epoch being probed in e and the membership of this epoch in M. The process initializes these variables when it obtains the information about the current epoch from the configuration service. To probe an epoch e, the process sends a PROBE(e_{new}, e) message to the members of its configuration, asking them to join the new epoch e_{new} (line 36). Upon receiving this message (line 46), a process first checks that the proposed epoch e_{new} is $\geq$ the highest epoch it has ever been asked to join, which is stored in new_epoch (we always have epoch $\leq$ new_epoch). In this case, the process sets new_epoch to e_{new}. Then, if the process was initialized at an epoch $\geq$ the epoch e being probed, it replies with PROBE_ACK($\text{TRUE}, e_{\text{new}}$); otherwise, it replies with PROBE_ACK($\text{FALSE}, e_{\text{new}}$).

If p_r receives at least one PROBE_ACK($\text{FALSE}, e_{\text{new}}$) from a member of e (line 37), p_r can conclude that e has not been activated, since one of its processes was not initialized by the leader of this epoch. The process p_r can also be sure that e will never become activated, since it has switched at least one of its members to the new epoch. In this case, p_r starts probing the preceding epoch $e - 1$. Since no application message could have been committed in e, picking a new leader from an earlier epoch will not lose any committed messages and thus will not violate Invariant 1. If p_r receives some PROBE_ACK($\text{TRUE}, e_{\text{new}}$) messages, then it ends probing: any process p_j that replied in this way can be selected as the new leader (in particular, p_r is free to maintain the old leader if this is one of the processes that replied).

Reconfiguration: initialization. Once the probing finds a new leader p_j (line 39), the process p_r computes the membership of the new configuration using a function compute_membership (line 40). We do not prescribe any particular implementation for this function, except that the new membership must contain the new leader p_j. In practice, the function would take into account the desired changes to be made by the reconfiguration. Once the new configuration is computed, p_r attempts to store it in the configuration service using a compare_and_swap operation. This succeeds if and only if the current epoch in the configuration service is still the epoch from which p_r started probing, which implies that no concurrent reconfiguration occurred during probing. In this case p_r sends a NEW_CONFIG message with the new configuration to the new leader and returns the new configuration to the caller of reconfigure; otherwise, it returns $\bot$. A successful compare_and_swap also generates an $\texttt{introduction}_r$ action for the new configuration, which is used in the broadcast specification (§3).

When the new leader receives the NEW_CONFIG message (line 53), it sets status to LEADER, epoch to the new epoch, and stores the information about the new configuration in members and leader. The leader also sets next to the first free slot in the msg array and saves its initial length in a variable init_len. The leader then invokes conf_changed for the new configuration. In order to finish the reconfiguration, the leader needs to transfer its state to the other members of the configuration. To this end, the leader sends a NEW_STATE message to them, which contains the new epoch and a copy of its msg array (line 63; a practical implementation would optimize this by sending to each process only the state it is missing). Upon receiving a NEW_STATE message (line 64), a process overwrites its msg array with the one provided by the leader, sets its status to FOLLOWER, epoch to the new epoch, and leader to the new leader. The process also invokes conf_changed for the new configuration. It then acknowledges its initialization to the leader with an NEW_STATE_ACK message. Upon receiving NEW_STATE_ACK messages from all followers (line 74), the new leader sends COMMITs for all application messages from the previous epoch, delimited by init_len. These messages can be safely delivered, since they are now stored by all members of epoch e.

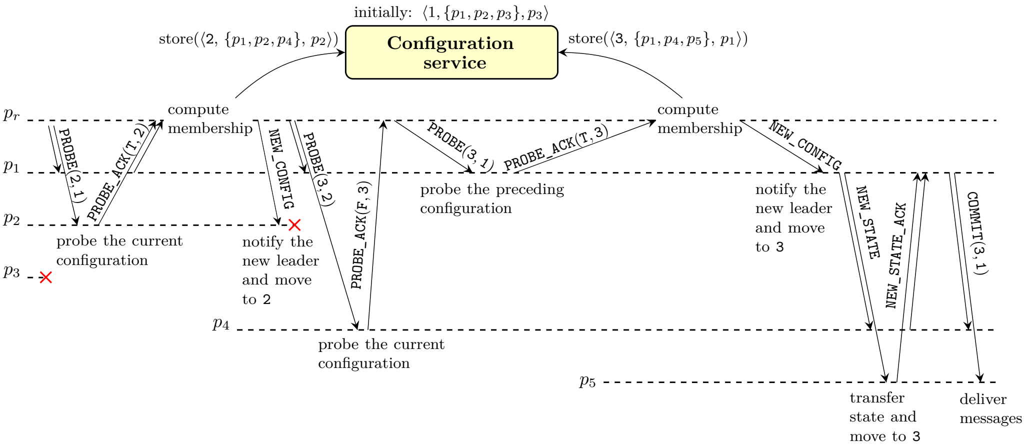

initially: $\langle 1, \{p_1, p_2, p_3\}, p_3\rangle$
store($\langle 2,\ \{p_1, p_2, p_4\},\ p_2\rangle$)
Configuration service
store($\langle 3,\ \{p_1, p_4, p_5\},\ p_1\rangle$)
p_r
p_1
p_2
p_3
p_4
p_5
PROBE(2,1)
PROBE_ACK(T,2)
compute membership
NEW_CONFIG
PROBE(3,2)
PROBE_ACK(F,3)
PROBE(3,1)
PROBE_ACK(T,3)
compute membership
NEW_CONFIG
probe the current configuration
notify the new leader and move to 2
probe the current configuration
probe the preceding configuration
notify the new leader and move to 3
NEW_STATE
NEW_STATE_ACK
COMMIT(3,1)
transfer state and move to 3
deliver messages

Figure 4 The behavior of the protocol during reconfiguration.

Example. Figure 4 gives an example illustrating the message flow of reconfiguration. Assume that the initial configuration 1 consists of processes p_1, p_2 and p_3. Following a failure of p_3, a process p_r initiates reconfiguration to move the system to a new configuration 2. To this end, p_r sends PROBE(2, 1) to the members of configuration 1. Both processes p_1 and p_2 respond to p_r with PROBE_ACK(TRUE, 2). The process p_r computes the membership of the new configuration, replacing p_3 by a fresh process p_4, and stores the new configuration in the configuration service, with p_2 as the new leader. Next, p_r sends a NEW_CONFIG message to p_2.

Assume that after receiving this message p_2 fails, prompting p_r to initiate yet another reconfiguration to move the system to a configuration 3. To this end, p_r sends PROBE(3, 2) to the members of configuration 2, and p_4 responds with PROBE_ACK(FALSE, 3). The process p_r concludes that epoch 2 has not been activated and starts probing the preceding epoch 1: it sends PROBE(3, 1) and gets a reply PROBE_ACK(TRUE, 3) from p_1, which is selected as the new leader. The process p_r computes the new set of members, replacing p_3 by a fresh process p_5, stores the new configuration in the configuration service, and sends a NEW_CONFIG message to the new leader p_1. This process invokes the conf_changed upcall for the new configuration and sends its state to the followers in a NEW_STATE message. The followers store the state, invoke conf_changed upcalls and reply with NEW_STATE_ACKs. Upon receiving these, p_1 sends COMMITs for all application messages in its state.

Steady-state latency and reconfiguration downtime. A configuration is *functional* if it was activated and all its members are correct. A configuration is *stable* if it is functional and no configuration with a higher epoch is introduced. The *steady-state latency* is the maximum number of message delays it takes from the moment the leader p_i of a stable configuration receives a broadcast request for a message m and until m is delivered by p_i. It is easy to see that our protocol has the steady-state latency of 2 (assuming self-addressed messages are received instantaneously), which is optimal [22].

The system may be reconfigured not only in response to a failure, but also to make changes to a functional configuration: e.g., to move replicas from highly loaded machines to lightly loaded ones, or to change the number of machines replicating the service [26,29,39]. As modern online services have stringent availability requirements, it is important to minimize the period of time when a service is unavailable due to an ongoing reconfiguration. More precisely, suppose the system is being reconfigured from a functional configuration C to a stable configuration C'. The reconfiguration *downtime* is the maximum number of message delays it takes from the moment C is disabled and until the leader of C' is ready to broadcast application messages in the new configuration.

As we argue in §7, existing vertical solutions for atomic broadcast stop the system as the first step of reconfiguration [3], resulting in the reconfiguration downtime of at least 4 (2 message delays to disable the latest functional configuration plus at least 2 message delays to reach consensus on the next configuration and propagate the decision). In contrast, our protocol achieves the downtime of 0 by keeping the latest functional configuration active while the probing of past configurations and agreement on a new one is in progress.

▶ **Theorem 1.** *The VAB protocol reconfigures a functional configuration with* 0 *downtime.*

Proof. Suppose that the current configuration C with an epoch e is functional. Note that the normal path of our protocol is guarded by preconditions $\mathsf{epoch} = e$, so that C can broadcast and deliver application messages as long as this holds at all its members (lines 19, 23 and 26). Assume now that a process p_r starts reconfiguring the system to a new configuration C' with epoch $e + 1$. The process p_r will send `PROBE` messages to the members of C and, since C is functional, p_r will only get replies $\texttt{PROBE\_ACK}(\text{TRUE}, e + 1)$. Handling a `PROBE` message only modifies the `new_epoch` variable, not `epoch`. Therefore, C can continue processing broadcasts while p_r is probing its members, storing C' in the configuration service, and sending $\texttt{NEW\_CONFIG}(e + 1, \_)$ to the leader p_i of C'. When the new leader p_i handles $\texttt{NEW\_CONFIG}(e + 1, \_)$, it will set $\mathsf{epoch} = e + 1$, disabling the old configuration. However, the leader will at once be ready to broadcast messages in the new configuration, as required. ◀

Correctness. Our protocol achieves the above 0-downtime guarantee without violating correctness. Informally, this is because it always chooses the leader of the new configuration from among the members of the latest activated configuration, and a message can only be delivered in this configuration after having been replicated to all its members. Hence, the new leader will immediately know about all previously delivered messages, including those delivered during preliminary reconfiguration steps. The following theorem (proved in [4, §A]) states the correctness of our protocol.

▶ **Theorem 2.** *The VAB protocol correctly implements reconfigurable atomic broadcast as defined in Figure 1.*

5 Passive Replication

The protocol presented in the previous section can be used to build reconfigurable fault-tolerant services via *active* (aka state-machine) replication [33]. Here a service is defined by a deterministic state machine and is executed on several replicas, each maintaining a copy of the machine. All replicas execute all client commands, which they receive via atomic broadcast. Together with the state machine's determinism, this ensures that each command yields the same result at all replicas, thus maintaining an illusion of a centralized fault-tolerant service.

In the rest of the paper, we focus on an alternative approach of building reconfigurable fault-tolerant services via *passive* (aka primary-backup) replication [6]. Here commands are only executed by the leader, which propagates the state changes induced by the commands to the other replicas. This allows replicating services with non-deterministic operations, e.g., those depending on timeouts or interrupts.

Formally, we consider services with a set of states $\mathcal{S}$ that accept a set of commands $\mathcal{C}$. A command $c \in \mathcal{C}$ can be executed using a call $\texttt{execute}(c)$, which produces its return value. Command execution may be non-deterministic. To deal with this, the effect of executing a command c on a state $\Sigma \in \mathcal{S}$ is defined by transition relation $\Sigma \xrightarrow{c} \langle r, \delta \rangle$, which produces a possible return value r of c and a *state update* δ performed by the command. The latter can be applied to any state Σ' using a function $\mathit{apply}(\Sigma', \delta)$, which produces a new state. For example, a command **if** $x = 0$ **then** $y \leftarrow 1$ **else** $y \leftarrow 0$ produces a state update $y \leftarrow 1$ when executed in a state with $x = 0$. A command assigning x to a random number may produce an update $x \leftarrow 42$ if the random generator returned 42 when the leader executed the command.

We would like to implement a service over a set of fault-prone replicas that is linearizable [16] with respect to a service that atomically executes commands on a single non-failing copy of the state machine. The latter applies each state update to the machine state Σ immediately after generating it, as shown in Figure 5. Informally, this means that commands appear to clients as if produced by a single copy of the state machine in Figure 5 in an order consistent with the *real-time order*, i.e., the order of non-overlapping command invocations.

5.1 Passive Replication vs Atomic Broadcast

As observed in [3,17,19], implementing passive replication requires propagating updates from the leader to the followers using a stronger primitive than atomic broadcast. To illustrate why, Figure 6 gives an incorrect attempt to simulate the specification in Figure 5 using our reconfigurable atomic broadcast (ignore the code in blue for now). This attempt serves as a strawman for a correct solution we present later. Each process keeps track of the epoch it belongs to in `cur_epoch` and the leader of this epoch in `cur_leader`. To execute a command (line 5), a process sends the command, tagged by a unique identifier, to the leader. It then waits until it hears back about the result.

A process keeps two copies of the service state – a *committed* state Σ and a *speculative* state Θ; the latter is only used when the process is the leader. When the leader receives a command c (line 10), it executes c on its speculative state Θ, producing a return value r and a state update δ. The leader immediately applies δ to Θ and distributes the triple of the command identifier, its return value and the state update via atomic broadcast. When a process (including the leader) delivers such a triple (line 15), it applies the update to its committed state Σ and sends the return value to the process the command originated at, determined from the command identifier. When a process receives a `conf_changed` upcall (line 18), it stores the information received in `cur_epoch` and `cur_leader`. If the process is the leader of the new epoch, it also initializes its speculative state Θ to the committed state Σ.

In passive replication, a state update is incremental with respect to the state it was generated in. Thus, to simulate the specification in Figure 5, it is crucial that the committed state Σ at a process delivering a state update (line 16) be the same as the speculative state Θ from which this state update was originally derived (line 12). This is captured by the following invariant. Let $\Sigma_i(k)$ denote the value of Σ at process p_i before the k-th action in the history (and similarly for Θ).

▶ **Invariant 2.** *Let h be a history of the algorithm in Figure 6. If $h_k = \mathtt{deliver}_i(m)$, then there exist j and $l < k$ such that $h_l = \mathtt{broadcast}_j(m)$ and $\Sigma_i(k) = \Theta_j(l)$.*

Unfortunately, if we use atomic broadcast to disseminate state updates in Figure 6, we may violate Invariant 2. We next present two examples showing how this happens and how this leads to violating linearizability. The examples consider a replicated counter x with two commands – an increment ($x \leftarrow x + 1$) and a read (**return** x). Initially $x = 0$, and then two clients execute two increments.

```
$\Sigma \leftarrow \Sigma_0 \in \mathcal{S}$
function execute($c$):
    $\Sigma \xrightarrow{c} \langle r, \delta \rangle$
    $\Sigma \leftarrow apply(\Sigma, \delta)$
    return $r$
```

Figure 5 Passive replication specification.

```
cur_epoch $\in \mathbb{N}$
cur_leader $\in \mathcal{P}$
$\Sigma \leftarrow \Sigma_0 \in \mathcal{S}$  // committed state
$\Theta \leftarrow \Theta_0 \in \mathcal{S}$  // speculative state
function execute($c$):
    $id \leftarrow$ get_unique_id()
    send EXECUTE($id, c$) to cur_leader
    wait until receive RESULT($id, r$)
    return $r$
when received EXECUTE($id, c$)
    pre: cur_leader $= p_i$
    $\Theta \xrightarrow{c} \langle r, \delta \rangle$
    $\Theta \leftarrow apply(\Theta, \delta)$
    broadcast($\langle id, r, \delta \rangle$)
upon deliver($\langle id, r, \delta \rangle$)
    $\Sigma \leftarrow apply(\Sigma, \delta)$
    send RESULT($id, r$) to $origin(id)$
upon conf_changed($\langle e, M, p_j \rangle$)
    // conf_changed($\langle e, M, p_j \rangle, \sigma$)
    cur_epoch $\leftarrow e$
    cur_leader $\leftarrow p_j$
    if $p_i = p_j$ then
        $\Theta \leftarrow \Sigma$
        // $\langle \_, \_, \delta_1 \rangle \dots \langle \_, \_, \delta_k \rangle \leftarrow \sigma$
        // forall $l = 1..k$ do $\Theta \leftarrow apply(\Theta, \delta_l)$
```

Figure 6 Passive replication on top of broadcast: code at process p_i.

Example 1. The two increments are executed by the same leader. The first one generates an update $\delta_1 = (x \leftarrow 1)$ and a speculative state $\Theta = 1$. Then the second generates $\delta_2 = (x \leftarrow 2)$. Atomic broadcast allows processes to deliver the updates in the reverse order, with δ_1 applied to a committed state $\Sigma = 2$. This violates Invariant 2. Assume now that after the increments complete we change the configuration to move the leader to a different process. This process will initialize its speculative state Θ to the committed state $\Sigma = 1$. If the new leader now receives a read command, it will return 1, violating the linearizability with respect to Figure 5.

Example 2. The first increment is executed by the leader of an epoch e, which generates $\delta_1 = (x \leftarrow 1)$. The second increment is executed by the leader of an epoch $e' > e$ before it delivers δ_1 and, thus, in a speculative state $\Theta = 0$. This generates $\delta_2 = (x \leftarrow 1)$. Finally, the leader of e' delivers δ_1 and then δ_2, with the latter applied to a committed state $\Sigma = 1$. This is allowed by atomic broadcast yet violates Invariant 2. It also violates linearizability similarly to Example 1: if now the leader of e' receives a read, it will incorrectly return 1.

5.2 Primary-Order Atomic Broadcast

To address the above problem, Junqueira et al. proposed *primary-order atomic broadcast (POabcast)* [18, 19], which strengthens the classical atomic broadcast. We now briefly review POabcast and highlight its drawbacks, which motivates an alternative proposal we present in the next section. In our framework we can define POabcast by adding the properties over histories h in Figure 7 to those of Figure 1. This yields a reconfigurable variant of POabcast that we call *reconfigurable primary-order atomic broadcast (RPOabcast)*. RPOabcast also modifies the interface of reconfigurable atomic broadcast (§3) by only allowing a process to call `broadcast` if it is the leader of its current configuration.

Property 6 (Local Order) restricts the delivery order of messages broadcast in the same epoch: they must be delivered in the order the leader broadcast them. Property 7 (Global Order) restricts the delivery order of messages broadcast in different epochs: they must be delivered in the order of the epochs they were broadcast in. Finally, Property 8 (Primary Integrity) ensures that the leader of an epoch e' does not miss relevant messages from previous epochs: each message broadcast in an epoch $e < e'$ either has to be delivered by the leader before entering e', or can never be delivered at all. Local and Global Order trivially imply Property 3 (Total Order), so we could omit it from the specification. POabcast is stronger than plain atomic broadcast: the latter can be implemented from the former if each process forwards messages to be broadcast to the leader of its configuration.

▶ **Proposition 3.** *Reconfigurable atomic broadcast can be implemented from RPOabcast.*

When the passive replication protocol in Figure 6 is used with POabcast instead of plain atomic broadcast, Invariant 2 holds, and the protocol yields a service linearizable with respect to the specification in Figure 5 [19]. In particular, Local Order disallows Example 1 from §5.1, and Primary Integrity disallows Example 2 (which does not violate either Local or Global Order). POabcast can be obtained from our Vertical Atomic Broadcast (VAB) algorithm in §4 as follows. First, VAB already guarantees both Local and Global Order: e.g., this is the case for Local Order because processes are connected by reliable FIFO channels.

▶ **Theorem 4.** *VAB guarantees Local and Global order.*

Second, to ensure Primary Integrity, neither the new leader nor the followers invoke `conf_changed` upon receiving `NEW_CONFIG` (line 61) or `NEW_STATE` (line 71). Instead, the leader first waits until it receives `NEW_STATE_ACK` messages from all followers (line 74) and tells the processes to deliver all application messages from the previous epoch via `COMMIT` messages. Only once a process delivers all these application messages does it invoke `conf_changed` for the new configuration (and if the process is the leader, starts broadcasting).

Deferring the invocation of `conf_changed` at the leader is the key to guarantee Primary Integrity. On the one hand, it ensures that, before the newly elected leader of an epoch e' generates `conf_changed`, it has delivered all application messages that could have been delivered in previous epochs: Invariant 1 from §4 guarantees that the leader's initial log includes all such messages. On the other hand, the leader can also be sure that any message broadcast in an epoch $< e'$ but not yet delivered can *never* be delivered by any process. This is because, by the time the leader generates `conf_changed`, all followers in e' have overwritten their log with that of the new leader.

Since deferring `conf_changed` results in deferring the start of broadcasting by the leader, the modified VAB protocol has a reconfiguration downtime of 2 messages delays. This cost is inherent: the lower bound of Friedman and van Renesse [13] on the latency of Strong Virtually Synchronous broadcast (a variant of POabcast) implies that any solution must have a non-zero downtime. In the next section we circumvent this limitation by introducing a weaker variant of POabcast, which we show sufficient for passive replication.

6 Speculative Primary-Order Atomic Broadcast

We now introduce *speculative primary-order atomic broadcast* (SPOabcast), a weaker variant of POabcast that allows implementing passive replication with minimal downtime. During reconfiguration, SPOabcast allows the new leader to deliver messages from previous epochs *speculatively* – without waiting for them to become durable – and start broadcast right away.

6. **Local Order.** If the leader of some epoch e receives $\texttt{broadcast}(m_1)$ before receiving $\texttt{broadcast}(m_2)$, then any process that delivers m_2 must also deliver m_1 before m_2:

$$\forall m_1, m_2, i, j, k, l, l'.\ h_k = \texttt{broadcast}_i(m_1) \ \wedge\ h_l = \texttt{broadcast}_i(m_2) \ \wedge\ k < l \ \wedge \\ \text{epochOf}(k) = \text{epochOf}(l) \ \wedge\ h_{l'} = \texttt{deliver}_j(m_2) \implies \exists k'.\ h_{k'} = \texttt{deliver}_j(m_1) \ \wedge\ k' < l'$$

7. **Global Order.** Assume the leaders of e and $e' > e$ receive $\texttt{broadcast}(m_1)$ and $\texttt{broadcast}(m_2)$ respectively. If a process p_i delivers m_1 and m_2, then it must deliver m_1 before m_2:

$$\forall m_1, m_2, i, k, k', l, l'.\ h_k = \texttt{broadcast}(m_1) \ \wedge\ h_l = \texttt{broadcast}(m_2) \ \wedge \\ \text{epochOf}(k) < \text{epochOf}(l) \ \wedge\ h_{k'} = \texttt{deliver}_i(m_1) \ \wedge\ h_{l'} = \texttt{deliver}_i(m_2) \implies k' < l'$$

8. **Primary Integrity.** Assume some process delivers an application message m originally broadcast in an epoch e. If any process p_i joins an epoch $e' > e$, then p_i must deliver m before joining e':

$$\forall m, i, k, l, l', e, e'.\, h_k = \texttt{broadcast}(m) \ \wedge\ \text{epochOf}(k) = e \ \wedge\ h_l = \texttt{deliver}(m) \ \wedge \\ h_{l'} = \texttt{conf\_changed}_i(\langle e', \_, \_\rangle) \ \wedge\ e < e' \implies \exists k'.\ h_{k'} = \texttt{deliver}_i(m) \ \wedge\ k' < l'$$

Figure 7 Properties of reconfigurable primary-order atomic broadcast over a history h.

9. **Basic Speculative Delivery Properties.** A process p_i can speculatively deliver a given application message m at most once in a given epoch and only if p_i is the leader of the epoch, m has previously been broadcast, and m has not yet been delivered by p_i:

$$\forall i, j, k, \sigma.\, h_k = \texttt{conf\_changed}_i(\langle \_, \_, p_j\rangle, \sigma) \implies (\sigma \neq \bot \implies p_i = p_j) \ \wedge\ (\forall m_1, m_2 \in \sigma.\, m_1 \neq m_2) \\ \wedge\ (\forall m \in \sigma.\ (\exists l.\ h_l = \texttt{broadcast}(m) \ \wedge\ l < k) \ \wedge\ (\neg\exists l.\ h_l = \texttt{deliver}_i(m) \ \wedge\ l < k))$$

10. **Prefix Consistency.**

 a. Consider m_1 and m_2 broadcast in different epochs. Assume that a process p_i delivers m_2, and a process p_j broadcasts m_2 in an epoch e'. Then p_i delivers m_1 before m_2 iff p_j delivers m_1 before joining e' or speculatively delivers m_1 when joining e':

$$\forall m_1, m_2, i, j, k_0, l_0, k, l, l', \sigma, e'.\ h_{k_0} = \texttt{broadcast}(m_1) \ \wedge\ h_{l_0} = \texttt{broadcast}(m_2) \ \wedge \\ \text{epochOf}(k_0) \neq \text{epochOf}(l_0) \ \wedge\ h_k = \texttt{deliver}_i(m_2) \ \wedge\ h_l = \texttt{broadcast}_j(m_2) \ \wedge \\ h_{l'} = \texttt{conf\_changed}_j(\langle e', \_, p_j\rangle, \sigma) \ \wedge\ \text{epochOf}(l) = e' \implies \\ ((\exists k'.\, h_{k'} = \texttt{deliver}_i(m_1) \wedge k' < k) \iff ((\exists l''.\, h_{l''} = \texttt{deliver}_j(m_1) \wedge l'' < l') \vee m_1 \in \sigma)$$

 b. Consider m_1 and m_2 broadcast in different epochs. Assume that a process p_i delivers m_2, and a process p_j speculatively delivers m_2 when joining an epoch e'. Then p_i delivers m_1 before m_2 iff p_j delivers m_1 before joining e' or speculatively delivers m_1 before m_2 when joining e':

$$\forall m_1, m_2, i, j, k_0, l_0, k, l, \sigma, e'.\ h_{k_0} = \texttt{broadcast}(m_1) \ \wedge\ h_{l_0} = \texttt{broadcast}(m_2) \ \wedge \\ \text{epochOf}(k_0) \neq \text{epochOf}(l_0) \ \wedge\ h_k = \texttt{deliver}_i(m_2) \ \wedge\ h_l = \texttt{conf\_changed}_j(\langle e', \_, p_j\rangle, \sigma) \ \wedge \\ m_2 \in \sigma \implies ((\exists k'.\ h_{k'} = \texttt{deliver}_i(m_1) \ \wedge\ k' < k) \iff \\ ((\exists l'.\ h_{l'} = \texttt{deliver}_j(m_1) \ \wedge\ l' < l) \ \vee\ \sigma = {\_}m_1{\_}m_2{\_}))$$

p_i	p_j
deliver m_1	(speculatively) deliver m_1
deliver m_2	broadcast m_2

(a)

time ↓

p_i	p_j
deliver m_1	(speculatively) deliver m_1
deliver m_2	speculatively deliver m_2

(b)

Figure 8 Properties of speculative primary-order atomic broadcast over a history h. Property 10 replaces Property 8 from Figure 7. The tables summarize its action orderings: the actions at the top happen before the actions at the bottom.

SPOabcast specification. SPOabcast modifies the interface of reconfigurable atomic broadcast (§3) in two ways. First, like in POabcast, a process can call `broadcast` only if it is the leader. Second, the `conf_changed` upcall for a configuration C carries an additional argument σ: `conf_changed`(C, σ). When the upcall is invoked at the leader of C, σ is a sequence of messages *speculatively delivered* to the leader (σ is not used at followers). SPOabcast is defined by replacing Primary Integrity in the definition of POabcast by the properties in Figure 8. Property 9 is self-explanatory. Property 10 (Prefix Consistency) constrains how speculative deliveries are ordered with respect to ordinary deliveries and broadcasts. For the ease of understanding, in Figure 8 we summarize these orderings in tables.

Part (a)/"only if" of Prefix Consistency is a weaker form of Primary Integrity. Assume that the leader p_j of an epoch e' broadcasts a message m_2. The property ensures that for any message m_1 delivered before m_2 at some process p_i, the leader p_j has to either deliver m_1 before joining e' or *speculatively deliver* m_1 *when joining* e'. As we demonstrate shortly, the latter option, absent in Primary Integrity, allows our implementation of SPOabcast to avoid extra downtime during reconfiguration. Part (a)/"if" conversely ensures that, if the leader p_j speculatively delivers m_1 before broadcasting m_2, then m_1 must always be delivered before m_2. This ensures that the speculation performed by the leader p_j is correct if any of the messages it broadcasts (e.g., m_2) are ever delivered at any process. Part (b) of Prefix Consistency ensures that the order of messages in a sequence speculatively delivered at a `conf_changed` upcall cannot contradict the order of ordinary delivery.

Speculative delivery provides weaker guarantees than ordinary delivery, since it does not imply durability. In particular, we allow a message to be speculatively delivered at a process p but never delivered anywhere, e.g., because p crashed. However, in this case Part (a)/"if" of Prefix Consistency ensures that all messages p broadcast after such a non-durable speculative delivery will also be lost. As we show next, this allows us to use SPOabcast to correctly implement passive replication without undermining its durability guarantees.

Passive replication using SPOabcast. The passive replication protocol in Figure 6 requires minimal changes to be used with SPOabcast, highlighted in blue. When the leader of an epoch e receives a `conf_changed` upcall for e (line 19), in addition to setting the speculative state Θ to the committed state Σ, the leader also applies the state updates speculatively delivered via `conf_changed` to Θ (lines 24-25). The leader can then immediately use the resulting speculative state to execute new commands (line 10). We prove the following in [4, §B].

▶ **Theorem 5.** *The version of the protocol in Figure 6 that uses SPOabcast satisfies Invariant 2 and implements a service linearizable with respect to the specification in Figure 5.*

In particular, part (a)/"only if" of Prefix Consistency disallows Example 2 from §5.1: it ensures that the leader broadcasting δ_2 will be aware of δ_1, either via ordinary or speculative delivery. More generally, part (a) ensures that, if a process p_i delivers a state update δ_2 broadcast by a leader p_j, then at the corresponding points in the execution, p_i and p_j are aware of the same set of updates (cf. the table in Figure 8). Part (b) of Prefix Consistency furthermore ensures that the two processes apply these updates in the same order. This contributes to validating Invariant 2 and, thus, the specification in Figure 5.

Implementing SPOabcast. To implement SPOabcast we modify the Vertical Broadcast Protocol in Figures 2-3 as follows. First, since `broadcast` can only be called at the leader, we replace lines 10-12 by line 13. Thus, the leader handles `broadcast` calls in the same way it previously handled FORWARD messages. Second, we augment conf_changed upcalls with speculative deliveries, replacing line 61 by line 62, and line 71 by line 72. Thus, the conf_changed upcall at the leader speculatively delivers all application messages in its log that have not yet been (non-speculatively) delivered. It is easy to check that these modifications do not change the 0-downtime guarantee of Vertical Atomic Broadcast.

▶ **Theorem 6.** *The primary-order version of the Vertical Atomic Broadcast protocol is a correct implementation of speculative primary-order atomic broadcast.*

Thus, Theorems 5 and 6 allow us to use VAB to replicate even non-deterministic services while minimizing the downtime from routine reconfigurations, e.g., those for load balancing.

We prove Theorem 6 in [4, §C]. Here we informally explain why the above protocol validates the key part (a)/"only if" of Prefix Consistency, weakening Primary Integrity (cf. the explanations we gave regarding the latter at the end of §5.2). On the one hand, as in the ordinary VAB, Invariant 1 from §4 guarantees that the log of a newly elected leader of an epoch e' contains all application messages m_1 that could have been delivered in epochs $< e'$. The new leader will either deliver or speculatively deliver all such messages before broadcasting anything (line 62). On the other hand, if the leader broadcasts a message m_2, then a follower will only accept it after having overwritten its log with the leader's initial one, received in NEW_STATE (line 64). This can be used to show that, if m_2 is ever delivered, then any message broadcast in an epoch $< e'$ that was not in NEW_STATE will never get delivered.

7 Related Work

The *vertical* paradigm of implementing reconfigurable services by delegating agreement on configuration changes to a separate component was first introduced by Lynch and Shvartsman [27] for emulating dynamic atomic registers. It was further applied by Lamport et al. [23] to solve reconfigurable single-shot consensus, yielding the Vertical Paxos family of protocols. Vertical Paxos and its follow-ups [3, 5, 11, 25, 28] require prior configurations to be disabled ("wedged") at the start of reconfiguration. In contrast, our VAB protocol allows the latest functional configuration to continue processing messages while the agreement on the next configuration is in progress. This results in the downtime of 0 when reconfiguring from a functional configuration. This feature is particularly desirable for atomic broadcast, where we want to keep producing new decisions when reconfiguration is triggered for load balancing rather than to handle failures.

To achieve the minimal downtime, the VAB protocol uses different epoch variables to guard the normal operation (epoch) and reconfiguration (new_epoch). By not modifying the epoch variable during the preliminary reconfiguration steps, the protocol allows the old configuration to operate normally while the reconfiguration is in progress (cf. the proof of Theorem 1 in §4). In contrast, Vertical Paxos uses a single epoch variable (maxBallot) for both purposes, thus disabling the current configuration at the start of reconfiguration. Our protocol for SPOabcast further extends the minimal downtime guarantee to the case of passive replication.

Both our VAB and SPOabcast protocols achieve an optimal steady-state latency of two message delays [22]. Although Junqueira et al. [19] show that no POabcast protocol can guarantee optimal steady-state latency if it relies on black-box consensus to order messages, our SPOabcast implementation is not subject to this impossibility result, as it does not use consensus in this manner.

Although the vertical approach has been widely used in practice [2, 8, 11, 32, 38], prior systems have mainly focused on engineering aspects of directly implementing a replicated state machine for a desired service rather than basing it on a generic atomic broadcast layer.

Our treatment of Vertical Atomic Broadcast develops a formal foundation that sheds light on the algorithmic core of these systems. This can be reused for designing future solutions that are provably correct and efficient.

Most reconfiguration algorithms that do not rely on an auxiliary configuration service can be traced back to the original technique of Paxos [21], which intersperses reconfigurations within the stream of normal command agreement instances. The examples of practical systems that follow this approach include SMART [26], Raft [31], and Zookeeper [35]. Other non-vertical algorithms [24] implement reconfiguration by spawning a separate non-reconfigurable state machine for each newly introduced configuration. In the absence of an auxiliary configuration service, these protocols require at least $2f + 1$ processes in each configuration [22], in contrast to $f + 1$ in our atomic broadcast protocols.

The fault-masking protocols of Birman et al. [3] and a recently proposed MongoDB reconfiguration protocol [34] separate the message log from the configuration state, but nevertheless replicate them at the same set of processes. As in non-vertical solutions, these algorithms require $2f+1$ replicas. They also follow the Vertical Paxos approach to implement reconfiguration, and as a result, may wedge the system prematurely as we explain above.

A variant of Primary Integrity, known as Strong Virtual Synchrony (or Sending View Delivery [9]), was originally proposed by Friedman and van Renesse [13] who also studied its inherent costs. Our SPOabcast abstraction is a relaxation of Strong Virtually Synchrony and primary-order atomic broadcast (POabcast) of Junqueira et al. [18,19]. Keidar and Dolev [20] proposed Consistent Object Replication Layer (COReL) in which every delivered message is assigned a color such that a message is "yellow" if it was received and acknowledged by a member of an operational quorum, and "green" if it was acknowledged by all members of an operational quorum. While the COReL's yellow messages are similar to our speculative messages, Keidar and Dolev did not consider their potential applications, in particular, their utility for minimizing the latency of passive replication.

References

1 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2), 2011. doi:10.1145/1944345.1944348.

2 Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4), 2013. doi:10.1145/2535930.

3 Kenneth Birman, Dahlia Malkhi, and Robbert van Renesse. Virtually synchronous methodology for building dynamic reliable services. In *Guide to Reliable Distributed Systems - Building High-Assurance Applications and Cloud-Hosted Services*, chapter 22. Springer, 2012.

4 Manuel Bravo, Gregory Chockler, Alexey Gotsman, Alejandro Naser-Pastoriza, and Christian Roldán. Vertical atomic broadcast and passive replication (extended version). *arXiv*, abs/2408.08702, 2024. URL: https://arxiv.org/abs/2408.08702.

5 Manuel Bravo and Alexey Gotsman. Reconfigurable atomic transaction commit. In *Symposium on Principles of Distributed Computing (PODC)*, 2019.

6 Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In *Distributed Systems (2nd Ed.)*. ACM Press/Addison-Wesley, 1993.

7 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2), 1996. doi:10.1145/226643.226647.

8 Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

9 Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4), 2001. doi:10.1145/503112.503113.

10 Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4), 2004. doi:10.1145/1041680.1041682.

11 Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Symposium on Operating Systems Principles (SOSP)*, 2015.

12 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2), 1988. doi:10.1145/42282.42283.

13 Roy Friedman and Robbert van Renesse. Strong and weak virtual synchrony in Horus. In *Symposium on Reliable Distributed Systems (SRDS)*, 1996.

14 Jason Gustafson. Hardening Kafka replication. Talk at Kafka Summit San Francisco, 2018. URL: https://www.confluent.io/kafka-summit-sf18/hardening-kafka-replication/.

15 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *International Conference on Distributed Computing Systems (ICDCS)*, 2003.

16 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990. doi:10.1145/78969.78972.

17 Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference (USENIX ATC)*, 2010.

18 Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Conference on Dependable Systems and Networks (DSN)*, 2011.

19 Flavio Paiva Junqueira and Marco Serafini. On barriers and the gap between active and passive replication. In *Symposium on Distributed Computing (DISC)*, 2013.

20 Idit Keidar and Danny Dolev. Efficient message ordering in dynamic networks. In *Symposium on Principles of Distributed Computing (PODC)*, 1996.

21 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998. doi:10.1145/279227.279229.

22 Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2), 2006. doi:10.1007/S00446-006-0155-X.

23 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and primary-backup replication. In *Symposium on Principles of Distributed Computing (PODC)*, 2009.

24 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1), 2010. doi:10.1145/1753171.1753191.

25 Leslie Lamport and Mike Massa. Cheap Paxos. In *Conference on Dependable Systems and Networks (DSN)*, 2004.

26 Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The SMART way to migrate replicated stateful services. In *European Conference on Computer Systems (EuroSys)*, 2006.

27 Nancy Lynch and Alex A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Symposium on Distributed Computing (DISC)*, 2002.

28 John MacCormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou, and Ryan Peterson. Niobe: A practical replication protocol. *ACM Trans. Storage*, 3(4), 2008. doi:10.1145/1326542.1326543.

29 Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka: The Definitive Guide.* O'Reilly Media, 2017.

30 Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*, 1988.

31 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

32 Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4(4), 2011. doi:10.14778/1938545.1938549.

33 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), 1990. doi:10.1145/98163.98167.

34 William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. Design and analysis of a logless dynamic reconfiguration protocol. In *Conference on Principles of Distributed Systems (OPODIS)*, 2021.

35 Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Paiva Junqueira. Dynamic reconfiguration of primary/backup clusters. In *USENIX Annual Technical Conference (USENIX ATC)*, 2012.

36 Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. In *Colloquium on Structural Information and Communication Complexity (SIROCCO)*, 2017.

37 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: Abstraction and optimal asynchronous solution. In *Symposium on Distributed Computing (DISC)*, 2017.

38 Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

39 Michael J. Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M. Hellerstein, Heidi Howard, Faisal Nawab, and Ion Stoica. Matchmaker Paxos: A reconfigurable consensus protocol. *J. Syst. Res.*, 1(1), 2021.

What Cannot Be Implemented on Weak Memory?

Armando Castañeda
Instituto de Matemáticas, Universidad Nacional Autónoma de México, Mexico

Gregory Chockler
Department of Computer Science, University of Surrey, Guildford, UK

Brijesh Dongol
Department of Computer Science, University of Surrey, Guildford, UK

Ori Lahav
School of Computer Science, Tel Aviv University, Israel

Abstract

We present a general methodology for establishing the impossibility of implementing certain concurrent objects on different (weak) memory models. The key idea behind our approach lies in characterizing memory models by their *mergeability properties*, identifying restrictions under which independent memory traces can be merged into a single valid memory trace. In turn, we show that the mergeability properties of the underlying memory model entail similar mergeability requirements on the specifications of objects that can be implemented on that memory model. We demonstrate the applicability of our approach to establish the impossibility of implementing standard distributed objects with different restrictions on memory traces on three memory models: strictly consistent memory, total store order, and release-acquire. These impossibility results allow us to identify tight and almost tight bounds for some objects, as well as new separation results between weak memory models, and between well-studied objects based on their implementability on weak memory models.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Concurrent algorithms

Keywords and phrases Impossibility, Weak Memory Models, Total-Store Order, Release-Acquire

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.11

Related Version *Full Version*: `https://arxiv.org/abs/2405.16611` [14]

Funding *Armando Castañeda*: Supported by DGAPA PAPIIT project IN108723 and Royal Society grant IES\R1\221226.
Gregory Chockler: Supported by Royal Society grant: IES\R1\221226; CHIST-ERA project REDONDA EP/Y036425/1; and EPSRC grants: EP/X037142/1 and EP/X015149/1.
Brijesh Dongol: Supported by VeTSS; Royal Society grant: IES\R1\221226; CHIST-ERA project REDONDA EP/Y036425/1; and EPSRC grants: EP/X037142/1, EP/X015149/1, EP/V038915/1, and EP/R025134/2.
Ori Lahav: Supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811) and the Israel Science Foundation (grant number 814/22).

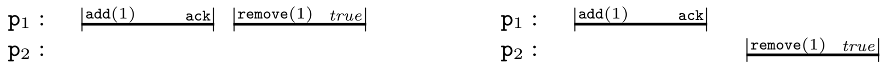
1 Introduction

Weak memory models have become standard in modern hardware architectures and programming languages. Unlike traditional strictly consistent memory (SCM), which provides *atomic* read/write instructions, memories achieve efficiency by multiple optimizations, which, in particular, delay propagation of writes instead of making them immediately visible to subsequent reads in other threads. Two well-studied models, which we consider in this paper,

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 11; pp. 11:1–11:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Example 1 Linearizable Obstruction-Free Set.

Consider a set object that provides the high-level operations `add`(v) and `remove`(v), where `remove` returns *true* iff the element v is in the abstract set and in this case removes v from the set. Consider the following histories assuming two processes:



history h_1 history h_2

Let σ_0 be the trace of a set implementation I generated by $\mathtt{p}_1$ executing `add`(1) until completion from the initial state, and for $i \in \{1, 2\}$, let σ_i be the trace generated by $\mathtt{p}_i$ after σ_0 to induce history h_i. Such traces must exist assuming I is obstruction-free. If σ_1 and σ_2 can be merged into a trace σ such that $\sigma_0 \cdot \sigma$ is a valid trace of a memory model M, then we reach a contradiction because $\mathtt{p}_1$ (resp., $\mathtt{p}_2$) cannot distinguish between σ_1 (resp., σ_2) and σ, and thus both `remove` operations of $\mathtt{p}_1$ and $\mathtt{p}_2$ in σ return *true*, contradicting linearizability of I. In other words, since the two `remove` invocations cannot be merged into a single linearizable object history, it must be that the corresponding memory traces cannot be merged. In particular, if σ_1 and σ_2 have neither RAW nor RMW, then they can always be merged on SCM, which gives us the impossibility result of [6] for this object.

are *total store order* model (TSO), as implemented in SPARC [36, 23] and x86 multiprocessors [31], and the weaker *release-acquire* model (RA), a fragment of C/C++11 [8, 26], which guarantees causal consistency together with per-location strict consistency (a.k.a. coherence).

The standard memory model for the design and analysis of asynchronous shared memory algorithms is SCM. These algorithms however, are not guaranteed to work correctly on weaker memory models (such as TSO and RA) due to the lack of atomicity of reads and writes. To ensure atomicity, one can use *fence* or atomic *read-modify-write (RMW)* instructions provided by the weak memory models. However, since fences and RMWs disable hardware optimizations and enforce synchronization between threads, they incur substantial performance overheads. Thus, one would like to understand when fences and RMWs are necessary and when they can be avoided, in order to correctly and efficiently implement the large body of existing shared memory algorithms on weak memory architectures.

In this paper, we set out to tackle this important and challenging question. The crux of our approach is based on *mergeability* of traces and object histories. Roughly speaking, two memory traces (sequences of memory accesses) of some memory model M are strongly (resp., weakly) mergeable if every (resp., some) interleaving of these traces forms a valid trace of M. Likewise, two object histories (sequences of invocations and responses) of some object O are strongly (resp., weakly mergeable) if every (resp., some) interleaving of these histories forms a valid history of O. Then, our key result is the *Merge Theorem*, which, roughly speaking, states that strongly (resp., weakly) mergeable memory traces can only be used to implement strongly (resp., weakly) mergeable object histories. Contrapositively, when operations of a certain concurrent object are not strongly (resp., weakly) mergeable, then the memory traces implementing these operations on a memory model M cannot be strongly (resp., weakly) mergeable in M. The correctness and progress conditions in the Merge Theorem are weaker versions of *linearizability* [22] and *obstruction-freedom* [21].

A prerequisite for applying our Merge Theorem for a particular memory model is to identify useful mergeability properties of the model. For SCM, TSO, and RA, we develop a set of properties (see Table 1) that describe conditions under which traces of the models can be (weakly/strongly) merged. These results provide key insights into the synchronization power of these memory models, and together with the Merge Theorem allow us to derive multiple impossibility results, and identify optimal implementations.

For instance, consider the read-after-write pattern (RAW), which is often used by shared memory algorithms under SCM (such as classical mutual exclusion [17, 28]) as a synchronization mechanism. In RAW, a process first writes to a shared variable and then reads from a different shared variable, and under SCM, this ensures that at least one of the two processes writing to two different variables has to observe the value written by the other process (see the SB program in §2). This means that solo traces that use RAW are not mergeable into a single trace. In turn, it is straightforward to establish that any two RAW-free read-write traces (by distinct processes) are weakly mergeable under SCM (§3).

With this observation, we easily re-establish (and generalize) the "Laws of Order" results from [6], showing that mutual exclusion protocols, as well as concurrent objects with *strongly non-commutative* methods, cannot be implemented on SCM with neither RAW nor RMW. We do so by simple mergeability-based arguments (see, e.g., Example 1), instead of rather complex and ad-hoc application of the covering technique used in [6]. Intuitively, two methods are strongly non-commutative if executing one of them first affects the response of the other, and vice versa. Moreover, by using mergeability properties for TSO and RA we directly obtain similar impossibility results for these models, whereas the argument in [6] for weak memory models is only implicit, based on the fact that enforcing a write to be executed before a read (i.e., implementing RAW) on a weak model requires a fence.

A benefit of our generic methodology is that we can also reason about implementability of methods that are not strongly non-commutative, hence not covered by [6]:

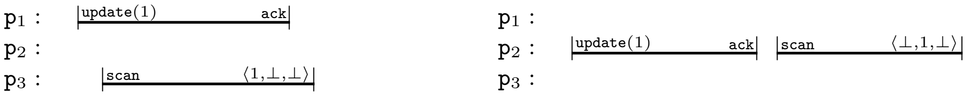
One-Sided Non-Commutative Operations. Some objects such as register, max-register, snapshot and monotone counter have pairs of methods that do not strongly non-commute. To support them, we consider *one-sided non-commutativity* of pairs of methods, which, roughly speaking, means that executing one of them first affects the response of the other, but not necessarily vice versa. We then apply the Merge Theorem to show that any linearizable obstruction-free implementations of these objects must use fences or RMWs in TSO and RA.

Then, for max-register, a useful building block in several implementations, e.g., [3, 7, 15], we obtain *fence-optimal* implementations in TSO and RA. The TSO implementation is obtained through a more general *fence-insertion strategy*: a transformation that takes any read/write linearizable implementation in SCM and adds fences between every write followed by a read or a return of an operation, provably resulting in a linearizable implementation in TSO. Combined with a wait-free read/write max-register implementation in SCM (with uses neither RAW nor RMW), the transformation gives a fence-optimal wait-free read/write max-register implementation in TSO. For RA, we develop a similar linearizable implementation by placing a fence in the beginning and the end of every operation, which leads to a fence-optimal implementation of max-register in RA.

Snapshot and Counter. We also reason about snapshot and (non-monotone) counter, which fall beyond the scope of non-commutativity. These two objects are of particular interests: snapshot is *universal* for a family of objects whose pairs of operations either commute or one overwrites the other [5], and counter is a useful building block for randomized consensus [2, 4]. For TSO, the fence-insertion transformation above once again provides a wait-free fence-optimal snapshot (resp., counter) implementation where every update operation ends with a fence. However, we use our Merge Theorem to show that, in sharp contrast to max-register, there is no obstruction-free read/write snapshot (resp., counter) implementation in RA, whose operations start with a fence and end with a fence (see outline in Example 2). To the best of our knowledge, this is the first sharp separation between max-register on the one hand and snapshot and counter on the other in terms of their implementability under RA using only reads, writes and fences.

Example 2 Linearizable Obstruction-Free Snapshot.

Mergeability can justify a novel impossibility result for RA, showing that a shared (single-writer multi-reader) snapshot object cannot be implemented with only reads, writes and fences under the restriction that all fences are only placed at the beginning and end of a method invocation. Consider the following histories assuming three processes:



history h_1 history h_2

An obstruction-free implementation should generate both histories. A merge-based argument implies that the memory traces σ_1 and σ_2 induced by the implementation when it generates h_1 and h_2 must not be mergable in the underlying memory model. Otherwise, the same algorithm will also allow some interleaving h of h_1 and h_2, but it is easy to observe that no such interleaving is linearizable: no valid single history h with only two updates, $\mathtt{update}(1)$ by $\mathtt{p}_1$ and $\mathtt{update}(1)$ by $\mathtt{p}_2$, can have *both* scan results $\langle 1, \bot, \bot\rangle$ and $\langle \bot, 1, \bot\rangle$. The RA memory model allows any two RMW-free traces σ_1 and σ_2 by disjoint sets of processes to be merged, provided that fences are not used in the middle of these traces. Roughly speaking, following [26, 24], the semantics of RA is based on point-to-point communication, making it is possible for $\mathtt{p}_1$ and $\mathtt{p}_3$ to communicate directly, without affecting $\mathtt{p}_2$. Thus, every implementation of snapshot on RA uses RMWs or fences in the middle of operations.

Outline. The rest of this paper is structured as follows. In §2 we define the notion of a memory model. In §3 we establish multiple mergeability properties for these memory models. In §4 we present the general impossibility result. In §5 we discuss applications of the theorem for well known objects, and tightness of the obtained lower bounds. We conclude and discuss related work in §6. The full version of that paper [14] contains more details and full proofs.

2 Weak Memory Models

In this paper, we consider three memory models:

Strictly Consistent Memory (SCM): In this model every write is propagated to all threads immediately after being executed. In the weak memory literature, this memory model is often referred to as sequential consistency, but it essentially corresponds to a collection of *linearizable* (a.k.a. atomic) register objects [22].

Total Store Order (TSO): Each process has a local FIFO store buffer. Writes are first enqueued in the buffer of the writing process, and later propagate from the buffer to main memory in an *internal* step that occurs non-deterministically as part of the system's execution. A read of a variable returns the latest write to the variable in the reading process' buffer or the value in main memory if there is no pending write to that variable in the buffer.

Release/Acquire (RA): This model employs a notion of synchronization between processes through acquiring instructions (read or RMW) which synchronize with previously executed releasing instructions (write or RMW) when the acquiring instruction reads its value from the releasing instruction. Such synchronization transfers "happens-before" knowledge from the releasing instruction to the acquiring instruction. Following a release-acquire synchronization, instructions that follow (in "happens-before" order) the acquire instruction must be consistent with the happens-before knowledge received through the synchronization.

The classic examples used to explain these memory models are the *store buffering* (SB), *independent reads of independent writes* (IRIW), and *message passing* (MP) programs, given below. We assume shared variables x and y initialized with the value 0 and process-local variables $a, b, \ldots$. The possible final values of $a, b, \ldots$ depend on the memory model.

Proc $\mathtt{p}_1$	Proc $\mathtt{p}_2$
$x := 1;$	$y := 1;$
$a := y;$	$b := x;$

(SB)

Proc $\mathtt{p}_1$	Proc $\mathtt{p}_2$	Proc $\mathtt{p}_3$	Proc $\mathtt{p}_4$
$x := 1;$	$a := x;$	$c := y;$	$y := 1;$
	$b := y;$	$d := x;$	

(IRIW)

Proc $\mathtt{p}_1$	Proc $\mathtt{p}_2$
$x := 1;$	$a := y;$
$y := 1;$	$b := x;$

(MP)

Under SCM, no execution of SB ends with $a = b = 0$, while this outcome is possible under both TSO and RA. Under both SCM and TSO, no execution of IRIW ends with $a = c = 1$ and $b = d = 0$, while this outcome is possible under RA, indicating that under RA, processes $\mathtt{p}_2$ and $\mathtt{p}_3$ observe the writes to x and y in a different order. In particular, under RA, suppose that both $\mathtt{p}_1$ and $\mathtt{p}_4$ execute their writes. It is possible for $\mathtt{p}_2$ (resp., $\mathtt{p}_3$) to read the new value for x (resp., y) then read the old value for y (resp., x). Although RA is weaker than both SCM and TSO, like TSO, RA maintains causal consistency as demonstrated MP. Under all three memory models, when MP terminates, if $a = 1$, then $b = 1$, indicating that if $\mathtt{p}_2$ is aware of the write to y by $\mathtt{p}_1$, then it must also be aware of the prior write to x.

Non-SCM-outcomes (a.k.a. *weak behaviors*) can be avoided in weak memory models by using *fence* instructions. In TSO fences drain the store buffer of the process that executes the fence. In RA fences synchronize *in pairs*, transferring happens-before knowledge from one process to another. We formally include fences also in SCM (with "no-op" semantics).

2.1 Formalizing Weak Memory Models

For the formal definitions of the models, we find it most convenient to follow an operational presentation, where memory models are specified by labeled transition systems.

Sequences. For a sequence $s = \langle x_1, \ldots, x_n \rangle$, $s[i]$ denotes the ith element of s (i.e., x_i), and $|s|$ denotes the length of s (i.e., n). We write $x \in s$ when $s[i] = x$ for some $1 \leq i \leq n$. We denote by ε the empty sequence, write $s_1 \cdot s_2$ for concatenation of s_1 and s_2 and denote by X^* the set of all sequences over elements of a set X. The restriction of a sequence s w.r.t. a set Y, denoted $s|_Y$, is the longest subsequence of s that consists only of elements in Y. These notations are lifted to sets in the obvious way (e.g., $S \cdot s' \triangleq \{s \cdot s' \mid s \in S\}$ and $S|_Y \triangleq \{s|_Y \mid s \in S\}$). We use the suffix '-set' to lift a function f from some set X to a function form sequences over X, formally defined by: $f\text{-}\mathsf{set}(s) \triangleq \{f(s[i]) \mid 1 \leq i \leq |s|\}$.

Labeled Transition Systems (LTSs). An LTS L consists of a set of states, $\mathsf{states}(L)$; an initial state, $\mathsf{init}(L) \in \mathsf{states}(L)$; a set of transition labels, $\mathsf{labels}(L)$; and a set of transitions, $\mathsf{trans}(L) \subseteq \mathsf{states}(L) \times \mathsf{labels}(L) \times \mathsf{states}(L)$. We write $q \xrightarrow{l}_L q'$ for $\langle q, l, q' \rangle \in \mathsf{trans}(L)$, and given $\pi = \langle l_1, \ldots, l_n \rangle \in \mathsf{labels}(L)^*$, we write $q \xrightarrow{\pi}_L q'$ for $\exists q_2, \ldots, q_n.\, q \xrightarrow{l_1}_L q_2 \xrightarrow{l_2}_L \ldots q_n \xrightarrow{l_n}_L q'$. An *execution fragment* of L is a sequence $\alpha = \langle q_0, l_1, q_1, l_2, \ldots, l_n, q_n \rangle$ of alternating states and transition labels such that $q_i \xrightarrow{l_{i+1}}_L q_{i+1}$ for every $0 \leq i \leq n-1$. The *trace* of α, denoted $\mathsf{trace}(\alpha)$, is the restriction of α w.r.t. $\mathsf{labels}(L)$. We denote by $\mathsf{traces}(L, q)$ the set of all sequences that are traces of some execution fragment α of L that starts from $q \in \mathsf{states}(L)$. An execution fragment α of L is an *execution* of L if it starts from $\mathsf{init}(L)$. A sequence π of transition labels is a *trace* of L if it is a trace of some execution of L. We denote by $\mathsf{traces}(L)$ the set of all traces of L (so we have $\mathsf{traces}(L) = \mathsf{traces}(L, \mathsf{init}(L))$).

Domains. We assume sets Var of *shared variables* and Val of *values* with a distinguished *initial* value $0 \in \mathsf{Val}$. We let $\mathsf{P} \triangleq \{\mathtt{p}_1, \ldots, \mathtt{p}_N\}$ be the set of process identifiers.

Memory Actions. Memory operations execute atomically using *memory actions*, which include both argument and return values. Formally, a *memory action* $a \in \mathsf{MemActs}$ is one the following (where $x \in \mathsf{Var}$ and $v, v_{\mathsf{old}}, v_{\mathsf{new}} \in \mathsf{Val}$): (i) write action of the form $\mathtt{W}(x, v)$; (ii) read action of the form $\mathtt{R}(x, v)$; (iii) RMW action of the form $\mathtt{RMW}(x, v_{\mathsf{old}}, v_{\mathsf{new}})$; and (iv) fence action of the form $\mathtt{F}$. We denote by $\mathsf{typ}(a)$ the type of the memory action a ($\mathtt{W}$, $\mathtt{R}$, $\mathtt{RMW}$, or $\mathtt{F}$) and by $\mathsf{var}(a)$ the variable accessed by action a (when applicable).

Memory Events. A *memory event* $e \in \mathsf{MemEvs}$ is a pair $e = p{:}a$ where $p \in \mathsf{P}$ and $a \in \mathsf{MemActs}$. We use $\mathsf{proc}(e)$ and $\mathsf{act}(e)$ to retrieve the components of e (p and a, respectively). The functions $\mathsf{typ}(\cdot)$ and $\mathsf{var}(\cdot)$ are lifted to events in the obvious way.

Memory Models. The semantics of the memory operations is given by an LTS, called a *memory model.* The transition labels of a memory model M, $\mathsf{labels}(M) \triangleq \mathsf{MemEvs} \cup \{\tau\}$, consist of memory events, as well as τ, which represents a silent memory internal step.

We demonstrate the formulation of TSO as an LTS. The formal models for SCM and RA can be found in [14].

▶ **Definition 2.1.** TSO's states are pairs $\langle m, b\rangle$, where $m \in \mathsf{Var} \to \mathsf{Val}$ is the main memory and $b \in \mathsf{P} \to (\mathsf{Var} \times \mathsf{Val})^*$ assigns a store buffer to every process; the initial state is $\mathsf{init}(\mathrm{TSO}) \triangleq \langle \lambda x.\, 0, \lambda p.\, \varepsilon\rangle$ (i.e., all variables in memory are zeroed and all store buffers are empty); and the transitions are as follows, where $\beta|_x$ denotes the restriction of a store buffer β to pairs of the form $\langle x, \_\rangle$:

$$\textsc{write}\quad \frac{e = p{:}\mathtt{W}(x, v) \qquad b' = b[p \mapsto b(p) \cdot \langle x, v\rangle]}{\langle m, b\rangle \xrightarrow{e} \langle m, b'\rangle}$$

$$\textsc{read-from-buffer}\quad \frac{e = p{:}\mathtt{R}(x, v) \qquad b(p)|_x = \_ \cdot \langle\langle x, v\rangle\rangle}{\langle m, b\rangle \xrightarrow{e} \langle m, b\rangle}$$

$$\textsc{read-from-memory}\quad \frac{e = p{:}\mathtt{R}(x, v) \qquad b(p)|_x = \varepsilon \qquad m(x) = v}{\langle m, b\rangle \xrightarrow{e} \langle m, b\rangle}$$

$$\textsc{rmw}\quad \frac{e = p{:}\mathtt{RMW}(x, v_{\mathsf{old}}, v_{\mathsf{new}}) \qquad b(p) = \varepsilon \qquad m(x) = v_{\mathsf{exp}}}{\langle m, b\rangle \xrightarrow{e} \langle m[x \mapsto v_{\mathsf{new}}], b\rangle}$$

$$\textsc{fence}\quad \frac{e = p{:}\mathtt{F} \qquad b(p) = \varepsilon}{\langle m, b\rangle \xrightarrow{e} \langle m, b\rangle}$$

$$\textsc{propagate}\quad \frac{b(p) = \langle\langle x, v\rangle\rangle \cdot \beta \qquad m' = m[x \mapsto v] \qquad b' = b[p \mapsto \beta]}{\langle m, b\rangle \xrightarrow{\tau} \langle m', b'\rangle}$$

Memory Sequences. We refer to sequences $\rho \in (\mathsf{MemEvs} \cup \{\tau\})^*$ as *memory sequences* and to sequences $\sigma \in \mathsf{MemEvs}^*$ as *observable memory sequences.* We use the following notations:

- $\sigma|_p$ denotes the restriction of σ w.r.t. $\{e \in \mathsf{MemEvs} \mid \mathsf{proc}(e) = p\}$.
- $\mathsf{otraces}(M, q)$ denotes the set of all observable memory sequences obtained by restricting traces of M from a state q to non-τ steps, i.e., $\mathsf{otraces}(M, q) \triangleq \mathsf{traces}(M, q)|_{\mathsf{MemEvs}}$.
- $\mathsf{otraces}(M) \triangleq \mathsf{traces}(M)|_{\mathsf{MemEvs}}$ is the set of all observable memory sequences of M.

Stable States. A state $q \in \mathsf{states}(M)$ is *stable* if $q \overset{\tau}{\not\to}_M q'$ for any $q' \in \mathsf{states}(M)$. Every state of SCM is stable, a state of TSO is stable iff all store buffers are empty, and a state of RA is stable iff all processes are aware of all writes.

Well-Behaved Memory Models. TSO is strictly weaker than SCM and RA is strictly weaker than TSO, which formally means that $\mathsf{otraces}(\mathrm{SCM}) \subsetneq \mathsf{otraces}(\mathrm{TSO}) \subsetneq \mathsf{otraces}(\mathrm{RA})$. In the sequel we will need the following assumption on memory models:

▶ **Definition 2.2.** A memory model M is *well-behaved* if there exists a simulation R from SCM to M whose codomain consists solely of stable states. That is, there should exist a relation $R \subseteq \mathsf{states}(\mathrm{SCM}) \times \{q \in \mathsf{states}(M) \mid q \text{ is stable}\}$ such that (i) $\langle \mathsf{init}(\mathrm{SCM}), \mathsf{init}(M)\rangle \in R$; and (ii) if $\langle m, q\rangle \in R$ and $m \xrightarrow{l}_{\mathrm{SCM}} m'$, then $q \xrightarrow{l}_M \xrightarrow{\tau}{}^*_M q'$ and $\langle m', q'\rangle \in R$ for some stable $q' \in \mathsf{states}(M)$.

Table 1 Merging observable memory sequences σ_1 and σ_2 such that $\mathsf{proc\text{-}set}(\sigma_1) \cap \mathsf{proc\text{-}set}(\sigma_2) = \emptyset$.

	Memory	Restrictions on σ_1			Restrictions on σ_2			Merge
#Name	model	process	events	pattern	process	events	pattern	property
1 TSO^{s}	TSO	solo	RW	—	solo	RW	—	Strong
2 RA^{s}_1	RA	—	RW	—	—	—	—	Strong
3 RA^{s}_2	RA	—	RWF	PPTF	—	—	PPTF	Strong
4 RA^{s}_3	RA	—	RWF	PPLF	—	—	PPLF	Strong
5 SCM^{w}	SCM	—	RW	RBW	—	—	—	Weak
6 TSO^{w}	TSO	solo	RWF	LTF	—	—	—	Weak
7 RA^{w}	RA	—	RWF	LTF	—	—	—	Weak

Note that if M is well-behaved, then $\sigma_0 \cdot \sigma \in \mathsf{otraces}(\text{SCM})$ implies that there exist a stable state $q \in \mathsf{states}(M)$ and a memory trace ρ_0 such that $\mathsf{init}(M) \xrightarrow{\rho_0}_M q$, $\rho_0|_{\mathsf{MemEvs}} = \sigma_0$, and $\sigma \in \mathsf{otraces}(M, q)$. The following lemma is proven in [14].

▶ **Lemma 2.3.** *Each* $M \in \{\text{SCM}, \text{TSO}, \text{RA}\}$ *is well-behaved.*

3 Mergeability Results for Memory Models

We consider two notions of mergeability of observable memory traces, *weak* mergeability, which means that *some* interleaving of the given traces is admitted, and *strong* mergeability, which requires that *all* interleavings are admitted. We denote by $s_1 \sqcup\!\sqcup\, s_2$ the the set of all interleavings of s_1 and s_2.

For our impossibility result to handle a non-empty base object history (as in Example 1), it does not suffice to merge memory traces from the initial state. Instead, we require the traces to be mergeable from every *stable* state:

▶ **Definition 3.1.** Two observable memory traces σ_1, σ_2 with $\mathsf{proc\text{-}set}(\sigma_1) \cap \mathsf{proc\text{-}set}(\sigma_2) = \emptyset$ are *weakly (resp., strongly) mergeable* in a memory model M if for every stable state $q_0 \in \mathsf{states}(M)$ such that $\sigma_1, \sigma_2 \in \mathsf{otraces}(M, q_0)$, we have $\sigma \in \mathsf{otraces}(M, q_0)$ for some (resp., every) $\sigma \in \sigma_1 \sqcup\!\sqcup\, \sigma_2$.

Table 1 presents the merge properties established for the memory models we consider (see [14] for the proofs). To specify restrictions on the mergeable traces, we say that an observable memory sequence σ is:

solo if $|\mathsf{proc\text{-}set}(\sigma)| = 1$;
***read-write* (*RW*)** if $\mathsf{typ\text{-}set}(\sigma) \subseteq \{\mathtt{R}, \mathtt{W}\}$;
***read-write-fence* (*RWF*)** if $\mathsf{typ\text{-}set}(\sigma) \subseteq \{\mathtt{R}, \mathtt{W}, \mathtt{F}\}$;
***read-before-write* (*RBW*)** if for every $k_1 < k_2$, if $\mathsf{typ}(\sigma[k_1]) = \mathtt{W}$, $\mathsf{typ}(\sigma[k_2]) = \mathtt{R}$, and $\mathsf{var}(\sigma[k_1]) \neq \mathsf{var}(\sigma[k_2])$, then $\mathsf{typ}(\sigma[k]) = \mathtt{W}$ and $\mathsf{var}(\sigma[k]) = \mathsf{var}(\sigma[k_2])$ for some $k_1 < k < k_2$;[1]
***trailing-fence* (*TF*)** if there is no k such that $\mathsf{typ}(\sigma[k]) = \mathtt{F}$ but $\mathsf{typ}(\sigma[k+1]) \neq \mathtt{F}$;
***leading-fence* (*LF*)** if there is no k such that $\mathsf{typ}(\sigma[k]) = \mathtt{F}$ but $\mathsf{typ}(\sigma[k-1]) \neq \mathtt{F}$;
***per-process trailing fence* (*PPTF*)** if $\sigma|_p$ is TF for all processes p;
***per-process leading fence* (*PPLF*)** if $\sigma|_p$ is LF for all processes p; and
***leading-and-trailing-fence* (*LTF*)** if $\sigma = \sigma_1 \cdot \sigma_2$ for some LF σ_1 and TF σ_2.

[1] RBW is equivalent to the absence of the read-after-write (RAW) pattern as defined in [6].

We have three types of restrictions, namely: (i) a restriction on the processes (solo); (ii) restrictions on the types of events (RW and RWF); and (iii) restrictions on the access pattern (all others). The restrictions on types and access patterns correspond to synchronization mechanisms that are expensive performance wise. RMWs and non-RBW were identified as such in [6], and since we explicitly deal with weak memory models, we add fences to this list. To motivate our focus on leading/trailing fence placement, we note that the trivial linearizable implementation of an atomic register using a write/read instruction requires fences: at the end of every write operation on TSO, and at the beginning and the end of every (write/read) operation on RA. We aim to investigate whether other objects admit similar implementations.

Next, we briefly discuss the results in the table:

SCM. In SCM, if σ_1 is RW-RBW, then it can be weakly merged with any other observable memory trace. Indeed, being RW-RBW, σ_1 must be of the form $\sigma_1^{\mathsf{r}} \cdot \sigma_1^{\mathsf{w}}$ where σ_1^{r} is a sequence of reads and σ_1^{w} is a sequence of writes and reads, starting with a write, where the reads in σ_1^{w} read from the writes in σ_1^{w}. Then, it is straightforward to see that σ_1 and any observable memory sequence σ_2 can be merged to form the trace $\sigma = \sigma_1^{\mathsf{r}} \cdot \sigma_2 \cdot \sigma_1^{\mathsf{w}}$, which is valid trace under SCM. We note that the RBW restriction is necessary here, as $\langle \mathtt{p}_1{:}\mathtt{W}(x,1), \mathtt{p}_1{:}\mathtt{R}(y,0)\rangle$ and $\langle \mathtt{p}_2{:}\mathtt{W}(y,1), \mathtt{p}_2{:}\mathtt{R}(x,0)\rangle$ (which may arise from the SB example) are not weakly mergeable. Also note that there is no useful *strong* merge property for SCM. Even $\langle \mathtt{p}_1{:}\mathtt{W}(x,1)\rangle$ and $\langle \mathtt{p}_2{:}\mathtt{R}(x,0)\rangle$ cannot be strongly merged.

TSO. In TSO, σ_1 and σ_2 can be *strongly* merged when they are both solo-RW traces. This holds because with only writes and reads, there is always an observable trace where *all* the writes of both σ_1 and σ_2 remain in the local store buffers, allowing the events of σ_1 and σ_2 to be arbitrarily interleaved. TSO also satisfies a weak merge property if σ_1 is solo-RWF-LTF and σ_2 is arbitrary. To do so, we let $\sigma_1 = \sigma_1^{\mathsf{lf}} \cdot \sigma_1' \cdot \sigma_1^{\mathsf{tf}}$ where $\mathsf{typ\text{-}set}(\sigma_1^{\mathsf{lf}}) \cup \mathsf{typ\text{-}set}(\sigma_1^{\mathsf{tf}}) \subseteq \{\mathtt{F}\}$ and σ_1' is RW. Then, $\sigma_1^{\mathsf{lf}} \cdot \sigma_1' \cdot \sigma_2 \cdot \sigma_1^{\mathsf{tf}}$ is a valid TSO observable trace since no instruction in σ_1' forces writes to propagate. We note that the solo restriction is essential. For example, $\langle \mathtt{p}_1{:}\mathtt{W}(x,1), \mathtt{p}_2{:}\mathtt{R}(x,1), \mathtt{p}_2{:}\mathtt{R}(y,0)\rangle$ and $\langle \mathtt{p}_4{:}\mathtt{W}(y,1), \mathtt{p}_3{:}\mathtt{R}(y,1), \mathtt{p}_3{:}\mathtt{R}(x,0)\rangle$ (which may arise from the IRIW example) are not weakly mergeable.

RA. We prove three strong merge properties for RA: ($\mathrm{RA}_1^{\mathsf{s}}$) If σ_1 is RW, then it can be strongly merged with σ_2 even when σ_1 is non-solo. Indeed, in the absence of RMWs and fences in σ_1, the writes in σ_1 can be propagated to other processes of σ_1, but never propagate to the processes of σ_2, and vice-versa. ($\mathrm{RA}_2^{\mathsf{s}}$) If σ_1 is RWF-PPTF and σ_2 is PPTF, the strong merge argument is as follows. First, we remove all the fences in σ_1, which results in an RW trace. From $\mathrm{RA}_1^{\mathsf{s}}$, this trace can be strongly merged with σ_2. In the resulting trace, we reintroduce the fences removed from σ_1 arbitrarily after the last read or write of the corresponding process. Regardless of whether this fence is before or after a fence of σ_2, the resulting fence synchronization has no effect since σ_2 is also PPTF. ($\mathrm{RA}_3^{\mathsf{s}}$) If σ_1 is RWF-PPLF and σ_2 is PPLF the argument is symmetric to $\mathrm{RA}_2^{\mathsf{s}}$. Finally, RA satisfies a weak merge property if σ_1 is RWF-LTF. As in the TSO weak merge property, we split $\sigma_1 = \sigma_1^{\mathsf{lf}} \cdot \sigma_1' \cdot \sigma_1^{\mathsf{tf}}$. By $\mathrm{RA}_1^{\mathsf{s}}$, $\sigma_1' \cdot \sigma_2$ is an RA observable trace. Then, $\sigma_1^{\mathsf{lf}} \cdot \sigma_1' \cdot \sigma_2 \cdot \sigma_1^{\mathsf{tf}}$ is an RA observable trace since the leading/trailing fences have no bearing on the execution.

4 A Recipe for Merge-Based Impossibility Results

We introduce objects, implementations, and histories (§4.1), and our main theorem (§4.2).

4.1 Objects and Their Implementations

We consider systems implementing of a high-level object O using the low-level atomic shared-memory operations provided by the memory model M.

Objects. An *object* O is a pair $O = \langle ops, rets \rangle$, where *ops* is a set of *operation names* (each of which may include argument values) and *rets* is a set of *response values*. We use $\mathsf{ops}(O)$ and $\mathsf{rets}(O)$ to retrieve the components of an object O (*ops* and *rets*, respectively). We use $\mathtt{ack}$ for a default response value for operations that do not return any value.

Object Actions. To delimit executions of operations of O, we use *object actions* that can be either *invocation actions* of the form $\mathtt{inv}(o)$ with $o \in \mathsf{ops}(O)$, or *response actions* of the form $\mathtt{res}(u)$ with $u \in \mathsf{rets}(O)$. We let $\mathsf{acts}(O)$ denote the set of all object actions of O.

Object Events. Like memory events defined in §2, *object events* are pairs $e = p{:}a$ where $p \in \mathsf{P}$ and $a \in \mathsf{acts}(O)$. We apply the same notations used for memory events to object events, and let $\mathsf{Evs}(O)$ denote the set of all object events. By *event* we collectively refer to either a memory event or an object event. Given a sequence π of events, we define the following notations:

- $\pi|_p$ denotes the restriction of π w.r.t. the set of events e with $\mathsf{proc}(e) = p$.
- $\pi|_{\mathsf{M}}$ denotes the restriction of π w.r.t. the set MemEvs of memory events.
- $\pi|_O$ denotes the restriction of π w.r.t. the set $\mathsf{Evs}(O)$ of object events.

Histories. A *history* of an object O is a sequence of events in $\mathsf{Evs}(O)$. We denote by $(p\colon |\underline{{}^{o}\quad{}^{u}}|)$ the history consisting of a single operation by process $p \in \mathsf{P}$ invoking $o \in \mathsf{ops}(O)$ with response value $u \in \mathsf{rets}(O)$ (and omit the response value if it is $\mathtt{ack}$), i.e., $(p\colon |\underline{{}^{o}\quad{}^{u}}|) \triangleq \langle p{:}\mathtt{inv}(o), p{:}\mathtt{res}(u) \rangle$ and $(p\colon |\underline{{}^{o}\qquad}|) \triangleq \langle p{:}\mathtt{inv}(o), p{:}\mathtt{res}(\mathtt{ack}) \rangle$. A history h is:

- *sequential* if it is a prefix of a history of the form $(p_1\colon |\underline{{}^{o_1}{}^{u_1}}|) \cdot (p_2\colon |\underline{{}^{o_2}{}^{u_2}}|) \cdots (p_n\colon |\underline{{}^{o_n}{}^{u_n}}|)$;
- *well-formed* if $h|_p$ is sequential for every $p \in \mathsf{P}$; and
- *complete* if it is well-formed and each $h|_p$ ends with a response event.

We let $\mathsf{H}(O)$, $\mathsf{ComH}(O)$, and $\mathsf{ComSeqH}(O)$ denote the sets of all well-formed histories of O, all complete histories of O, and all complete sequential histories of O (respectively).

Specifications. We assume that every object O is associated with a *specification*, denoted $\mathsf{spec}(O)$, that is a subset of $\mathsf{ComSeqH}(O)$ that is prefix-closed (in the sense that $h' \in \mathsf{spec}(O)$ for every $h' \in \mathsf{ComSeqH}(O)$ that is a prefix of some $h \in \mathsf{spec}(O)$). An object O is *deterministic* if no two histories in $\mathsf{spec}(O)$ have longest common prefix that ends with an invocation.

Implementations. An *implementation I of an operation o for a process p* is an LTS whose set of transition labels are events with process identifier p. We assume that a response event is always the last transition of executions of I (i.e., if $q \xrightarrow{p:\mathtt{res}(u)}_I q'$, then no transition is enabled in q'). An *implementation $\mathcal{I}$ of an object O* is a function assigning an implementation $\mathcal{I}(o, p)$ of o for p to every $o \in \mathsf{ops}(O)$ and $p \in \mathsf{P}$.

An implementation $\mathcal{I}$ of an object O induces an LTS, denoted $\mathsf{S}_{\mathcal{I}}$, that repeatedly and concurrently executes the operations of O as $\mathcal{I}$ prescribes. To formally define $\mathsf{S}_{\mathcal{I}}$, we first define the "per-process" LTS induced by $\mathcal{I}$, denoted $\mathsf{S}^p_{\mathcal{I}}$. This LTS is given by: $\mathsf{states}(\mathsf{S}^p_{\mathcal{I}}) \triangleq \{\bot\} \cup \{\langle o, q \rangle \mid o \in \mathsf{ops}(O), q \in \mathsf{states}(\mathcal{I}(o, p))\}$; $\mathsf{init}(\mathsf{S}^p_{\mathcal{I}}) \triangleq \bot$; $\mathsf{labels}(\mathsf{S}^p_{\mathcal{I}}) \triangleq \mathsf{Evs}(O) \cup \mathsf{MemEvs}$; and the transitions are given in Fig. 1. The state $\bot$ means that the

$$\frac{\begin{array}{c} e = p{:}\mathtt{inv}(o) \\ q = \mathsf{init}(\mathcal{I}(o,p)) \end{array}}{\bot \xrightarrow{e}_{\mathsf{S}^p_{\mathcal{I}}} \langle o, q \rangle} \qquad \frac{\begin{array}{c} e \in \mathsf{MemEvs} \\ q \xrightarrow{e}_{\mathcal{I}(o,p)} q' \end{array}}{\langle o, q \rangle \xrightarrow{e}_{\mathsf{S}^p_{\mathcal{I}}} \langle o, q' \rangle} \qquad \frac{\begin{array}{c} e = p{:}\mathtt{res}(u) \\ q \xrightarrow{e}_{\mathcal{I}(o,p)} - \end{array}}{\langle o, q \rangle \xrightarrow{e}_{\mathsf{S}^p_{\mathcal{I}}} \bot}$$

Figure 1 Transitions of $\mathsf{S}^p_{\mathcal{I}}$.

$$\frac{\bar{q}(p) \xrightarrow{e}_{\mathsf{S}^p_{\mathcal{I}}} q'}{\bar{q} \xrightarrow{e}_{\mathsf{S}_{\mathcal{I}}} \bar{q}[p \mapsto q']}$$

Figure 2 Transitions of $\mathsf{S}_{\mathcal{I}}$.

process is not currently executing any operation, whereas $\langle o, q \rangle$ means that process p is currently executing o and it is in state q of the implementation of o for p.

In turn, $\mathsf{S}_{\mathcal{I}}$ is given by: $\mathsf{states}(\mathsf{S}_{\mathcal{I}})$ is the set of all mappings assigning a state in $\mathsf{states}(\mathsf{S}^p_{\mathcal{I}})$ to every $p \in \mathsf{P}$; $\mathsf{init}(\mathsf{S}_{\mathcal{I}}) \triangleq \lambda p. \bot$; $\mathsf{labels}(\mathsf{S}_{\mathcal{I}}) \triangleq \mathsf{Evs}(O) \cup \mathsf{MemEvs}$; and the transition relation in Fig. 2. This transition simply interleaves the transitions of the different processes. In the sequel, we let $\mathsf{traces}(\mathcal{I}) \triangleq \mathsf{traces}(\mathsf{S}_{\mathcal{I}})$.

Histories of Implementations. Let $\mathcal{I}$ be an implementation of an object O, π_0 be a sequence of events, and M be a memory model. A history h of O is:

- *generated by* $\mathcal{I}$ *after* π_0 if $h = \pi|_O$ for some π such that $\pi_0 \cdot \pi \in \mathsf{traces}(\mathcal{I})$.
- *generated by* $\mathcal{I}$ *after* π_0 *under* M if $h = \pi|_O$ for some π such that $\pi_0 \cdot \pi \in \mathsf{traces}(\mathcal{I})$ and $(\pi_0 \cdot \pi)|_{\mathsf{M}} \in \mathsf{otraces}(M)$.

We denote by $\mathsf{H}(\pi_0, \mathcal{I})$ the set of all histories that are generated by $\mathcal{I}$ after π_0, and by $\mathsf{H}(\pi_0, \mathcal{I}, M)$ the set of all histories generated by $\mathcal{I}$ after π_0 under M. We also write $\mathsf{H}(\mathcal{I})$ instead of $\mathsf{H}(\varepsilon, \mathcal{I})$ and $\mathsf{H}(\mathcal{I}, M)$ instead of $\mathsf{H}(\varepsilon, \mathcal{I}, M)$.

4.2 The Merge Theorem

Our main result relates mergeability properties of memory models and objects implemented in those models, assuming that the implementation provides minimal safety and liveness guarantees. This result can be also seen as a *CAP Theorem* for weak memory models [19], where partition tolerance of CAP corresponds to mergeability, as it allows two traces of distinct set of processes to run concurrently without interaction. Our results are more fine grained, as we show the correspondence between mergeability of certain traces in a memory model, and the (in)ability of these traces to implement non-mergeable object histories.

For the formal treatment, we first present the following lemma (proven in [14]). The lemma describes the key shape of our results, namely that given two traces of an implementation over a memory model, the merge property over these traces carries over to a merge property over the histories induced by the traces.

▶ **Lemma 4.1.** *Let $\mathcal{I}$ be an implementation of O. Suppose that there exist sequences π_0, π_1, π_2 of events such that the following hold:*

(a) $\mathsf{proc\text{-}set}(\pi_1) \cap \mathsf{proc\text{-}set}(\pi_2) = \emptyset$*;* $\pi_0 \cdot \pi_1, \pi_0 \cdot \pi_2 \in \mathsf{traces}(\mathcal{I})$*;* $\pi_0|_O \in \mathsf{ComH}(O)$*; and*

(b) $\pi_0|_{\mathsf{M}} \cdot \sigma \in \mathsf{otraces}(M)$ *for some (resp., every)* $\sigma \in \pi_1|_{\mathsf{M}} \sqcup \pi_2|_{\mathsf{M}}$.

Then, $h \in \mathsf{H}(\pi_0, \mathcal{I}, M)$ for some (resp., every) $h \in \pi_1|_O \sqcup \pi_2|_O$.

The Merge Theorem, which we obtain using this lemma, makes several assumptions on implementations. First, the safety condition, which we call *consistency*, is restriction of linearizability to complete histories. For its definition, we first define reorderings of sequences.

▶ **Definition 4.2.** Let $R \subseteq X \times X$. A sequence $s' \in X^*$ is an *R-reordering* of a sequence $s \in X^*$ if there exists a bijection $f : \{1, \dots, |s|\} \to \{1, \dots, |s'|\}$ such that $s[i] = s'[f(i)]$ for every $1 \le i \le |s|$, and $f(i) < f(j)$ whenever $i < j$ and $\langle s[i], s[j] \rangle \in R$. We denote by $\mathsf{reorder}_R(s)$ the set of all R-reorderings of s, and lift this notation to sets by letting $\mathsf{reorder}_R(S) \triangleq \bigcup_{s \in S} \mathsf{reorder}_R(s)$.

We define sproc and lin relations on events:

$$\mathsf{sproc} \triangleq \{\langle e_1, e_2\rangle \mid \mathsf{proc}(e_1) = \mathsf{proc}(e_2)\}$$
$$\mathsf{lin} \triangleq \mathsf{sproc} \cup (\{e \mid e \text{ is a response event}\} \times \{e \mid e \text{ is a invocation event}\})$$

▶ **Definition 4.3.** A history $h' \in \mathsf{H}(O)$ *linearizes* a history $h \in \mathsf{H}(O)$, denoted $h \sqsubseteq h'$, if $h' \in \mathsf{reorder}_{\mathsf{lin}}(h)$. For a set $H' \subseteq \mathsf{H}(O)$, we write $h \sqsubseteq H'$ if $h \sqsubseteq h'$ for some $h' \in H'$.

▶ **Definition 4.4.** An implementation $\mathcal{I}$ of an object O is *consistent* under a memory model M if $h \sqsubseteq \mathsf{spec}(O)$ for every complete history $h \in \mathsf{H}(\mathcal{I}, M)$.

Consistency follows from linearizability [22], and it is equivalent to linearizability for implementations in which every history can be extended to a complete history.

Next, the liveness condition, which we call *availability*, requires progress for the specific histories under consideration.

▶ **Definition 4.5.** An implementation $\mathcal{I}$ of O is *available after a history* $h_0 \in \mathsf{ComSeqH}(O)$ w.r.t. a history $h \in \mathsf{H}(O)$ if $h \in \mathsf{H}(\pi_0, \mathcal{I}, \text{SCM})$ for every $\pi_0 \in \mathsf{traces}(\mathcal{I})$ such that $\pi_0|_{\mathsf{M}} \in \mathsf{traces}(\text{SCM})$ and $\pi_0|_O = h_0$. We say $\mathcal{I}$ is *available w.r.t. h*, if it is available after ε w.r.t. h (i.e., $h \in \mathsf{H}(\mathcal{I}, \text{SCM})$). We call $\mathcal{I}$ *spec-available* if for every $h_0, h \in \mathsf{ComSeqH}(O)$ such that $h_0 \cdot h \in \mathsf{spec}(O)$, $\mathcal{I}$ is available after h_0 w.r.t. h.

Availability w.r.t. h after h_0 only guarantees that the implementation under SCM is able to generate the history h when it starts executing after generating h_0. For deterministic implementations, availability w.r.t. h after h_0 follows from availability w.r.t. $h_0 \cdot h$ (after ϵ). Note that availability considers SCM rather than a general memory model M, but when M is well-behaved (Def. 2.2), $h \in \mathsf{H}(\pi_0, \mathcal{I}, \text{SCM})$ ensures that $h \in \mathsf{H}(\pi_0, \mathcal{I}, M)$. Spec-availability essentially means that the implementation can generate all (sequential) specification histories and for deterministic objects and implementations, it follows from obstruction-freedom [21].

The next lemma (proven in [14]) is used in the sequel to derive availability w.r.t. a history h from the fact that availability holds w.r.t. a sequential history that linearizes h.

▶ **Lemma 4.6.** *Suppose that $\mathcal{I}$ is available after h_0 w.r.t. a history $h' \in \mathsf{H}(O)$. Then, $\mathcal{I}$ is available after h_0 w.r.t. every $h \in \mathsf{H}(O)$ such that $h \sqsubseteq h'$.*

Next, we define mergeability for objects, akin to mergeability for memory models (Def. 3.1):

▶ **Definition 4.7.** Two histories $h_1, h_2 \in \mathsf{ComH}(O)$ with $\mathsf{proc\text{-}set}(h_1) \cap \mathsf{proc\text{-}set}(h_2) = \emptyset$ are *weakly (resp., strongly) mergeable in* $\mathsf{spec}(O)$ *after a history* $h_0 \in \mathsf{ComSeqH}(O)$ if $h_0 \cdot h_1 \sqsubseteq \mathsf{spec}(O)$ and $h_0 \cdot h_2 \sqsubseteq \mathsf{spec}(O)$ imply that $h_0 \cdot h \sqsubseteq \mathsf{spec}(O)$ for some (resp., every) $h \in h_1 \sqcup h_2$.

We now have all prerequisites to state our Merge Theorem (see [14] for the proof).

▶ **Theorem 4.8.** *Let $\mathcal{I}$ be an implementation of an object O that is consistent under a well-behaved memory model M. Suppose that there exist $\pi_0 \in \mathsf{traces}(\mathcal{I})$, $h_1, h_2 \in \mathsf{ComH}(O)$ such that the following hold, where $h_0 = \pi_0|_O$ and $\sigma_0 = \pi_0|_{\mathsf{M}}$:*

(i) $h_0 \in \mathsf{spec}(O)$, $\sigma_0 \in \mathsf{traces}(\text{SCM})$, *and* $\mathsf{proc\text{-}set}(h_1) \cap \mathsf{proc\text{-}set}(h_2) = \emptyset$,

(ii) *$\mathcal{I}$ is available after h_0 w.r.t. some $h^i_{\mathsf{seq}} \in \mathsf{ComSeqH}(O)$ such that $h_i \sqsubseteq h^i_{\mathsf{seq}}$ for $i \in \{1, 2\}$,*

(iii) *h_1 and h_2 are not weakly (resp., strongly) mergeable in* $\mathsf{spec}(O)$ *after h_0.*

Then, there exist π_1 and π_2 such that all of the following hold:

(a) *For* $i \in \{1,2\}$*, we have* $\pi_0 \cdot \pi_i \in \mathsf{traces}(\mathcal{I})$*;* $\pi_i|_O = h_i$*;* $\sigma_0 \cdot \pi_i|_{\mathsf{M}} \in \mathsf{traces}(\text{SCM})$*; and* $\mathsf{proc\text{-}set}(\pi_i) = \mathsf{proc\text{-}set}(h_i)$*.*

(b) *For every* $\pi'_1 \in \mathsf{reorder}_{\mathsf{sproc}}(\pi_1)$ *and* $\pi'_2 \in \mathsf{reorder}_{\mathsf{sproc}}(\pi_2)$ *such that* $\pi'_1|_O = h_1$*,* $\pi'_2|_O = h_2$*, and* $\sigma_0 \cdot \pi'_1|_{\mathsf{M}}, \sigma_0 \cdot \pi'_2|_{\mathsf{M}} \in \mathsf{traces}(\text{SCM})$*, we have that* $\pi'_1|_{\mathsf{M}}$ *and* $\pi'_2|_{\mathsf{M}}$ *are not weakly (resp., strongly) mergeable in* M*. In particular,* $\pi_1|_{\mathsf{M}}$ *and* $\pi_2|_{\mathsf{M}}$ *are not weakly (resp., strongly) mergeable in* M*.*

For simplicity, we explain Thm. 4.8 for $\pi_0 = \varepsilon$ (and hence $h_0 = \sigma_0 = \varepsilon$). The theorem assumes that we start with an implementation $\mathcal{I}$ that is consistent under the memory model M under consideration. Moreover, we assume that we have two complete histories h_1 and h_2 of the object such that the processes of h_1 and h_2 are distinct (condition (i)), $\mathcal{I}$ is available w.r.t. some linearization of h_1 and h_2 (condition (ii)), and that h_1 and h_2 are ***not*** weakly (strongly) mergeable (condition (iii)). Then, for $i \in \{1,2\}$ there must be a trace π_i of $\mathcal{I}$, corresponding to h_i, whose memory events are allowed by SCM, and processes are only included in π_i if they call some operation of the object (condition (a)), such that π_1 and π_2 restricted to memory events are ***not*** weakly (strongly) mergeable in M (second clause of condition (b)). In fact, weak (strong) ***non***-mergeability extends to any process-preserving reordering of π_1 and π_2 whose corresponding histories are h_1 and h_2 and corresponding memory traces are SCM traces (first clause of condition (b)).

5 Implementability of Objects on Weak Memory Models

We demonstrate the power of the Merge Theorem by using it along with the mergeability results in Table 1 to characterize implementability of objects under weak memory models.

5.1 One-Sided Non-Commutative Operations

We start by analyzing implementability of pair of operations o_1 and o_2 such that o_1 is *one-sided non-commutative* w.r.t. o_2. Roughly, this means that the execution order of o_1 and o_2 affects the response of o_1. Formally:

▶ **Definition 5.1.** An operation $o_1 \in \mathsf{ops}(O)$ is *one-sided non-commutative w.r.t. an operation* $o_2 \in \mathsf{ops}(O)$ *in* $\mathsf{spec}(O)$ if there exist $h_0 \in \mathsf{ComSeqH}(O)$, processes $p_1 \neq p_2$, and response values $u_1, v_1, u_2 \in \mathsf{rets}(O)$ such that: (i) $u_1 \neq v_1$; (ii) $h_0 \cdot (p_1\colon |\underline{o_1 u_1}|\,) \in \mathsf{spec}(O)$; and (iii) $h_0 \cdot (p_2\colon |\underline{o_2 u_2}|\,) \cdot (p_1\colon |\underline{o_1 v_1}|\,) \in \mathsf{spec}(O)$.

▶ **Example 5.2.** Consider a standard register object Reg with initial value 0, and operations $\texttt{write}(v)$, where $v \in V$ for some set of values V, and $\texttt{read}$. Then, $\texttt{read}$ is one-sided non-commutative w.r.t. $\texttt{write}$ in $\mathsf{spec}(\mathsf{Reg})$. Indeed, for $p_1 \neq p_2$ and $h_0 = \varepsilon$, we have both $(p_1\colon |\underline{\texttt{read}\;\;0}|\,) \in \mathsf{spec}(\mathsf{Reg})$ and $(p_2\colon |\underline{\texttt{write(1)}\;\;\;}|\,) \cdot (p_1\colon |\underline{\texttt{read}\;\;1}|\,) \in \mathsf{spec}(\mathsf{Reg})$. The same holds for *max-register* [3], denoted MaxReg, that stores integers with the initial value 0. We note that all pairs of specification histories of Reg and MaxReg with disjoint sets of processes are weakly mergeable.

▶ **Example 5.3.** Consider a monotone counter object MC with initial value 0, and operations $\texttt{inc}$ and $\texttt{read}$. Then, $\texttt{read}$ is one-sided non-commutative w.r.t. $\texttt{inc}$ in $\mathsf{spec}(\mathsf{MC})$ as for $p_1 \neq p_2$ and $h_0 = \varepsilon$, we have $(p_1\colon |\underline{\texttt{read}\;\;0}|\,) \in \mathsf{spec}(\mathsf{MC})$ and $(p_2\colon |\underline{\texttt{inc}\;\;}|\,) \cdot (p_1\colon |\underline{\texttt{read}\;\;1}|\,) \in \mathsf{spec}(\mathsf{MC})$.

The next lemma (proven in [14]) shows that for deterministic objects, the existence of a pair of operations one of which is one-sided non-commutative w.r.t. to the other implies that their corresponding histories are not strongly mergeable:

▶ **Lemma 5.4.** *Let O be a deterministic object and suppose that $o_1 \in \mathsf{ops}(O)$ is one-sided non-commutative w.r.t. $o_2 \in \mathsf{ops}(O)$ in $\mathsf{spec}(O)$. Then, there exist $h_0 \in \mathsf{ComSeqH}(O)$, processes $p_1 \neq p_2$, and response values $u_1, u_2 \in \mathsf{rets}(O)$ such that $(p_1\colon \lfloor o_1 u_1 \rfloor)$ and $(p_2\colon \lfloor o_2 u_2 \rfloor)$ are not strongly mergeable in $\mathsf{spec}(O)$ after h_0.*

Then, the following theorem (proven in [14]) follows from Thm. 4.8 and properties $\mathrm{TSO}^{\mathsf{s}}$ and $\mathrm{RA}_1^{\mathsf{s}}$ in Table 1.

▶ **Theorem 5.5.** *Let O be a deterministic object and suppose that $o_1 \in \mathsf{ops}(O)$ is one-sided non-commutative w.r.t. $o_2 \in \mathsf{ops}(O)$ in $\mathsf{spec}(O)$. Let $\mathcal{I}$ be a spec-available implementation of O that is consistent under $M \in \{\mathrm{TSO}, \mathrm{RA}\}$. Then, there exist $p_1, p_2 \in \mathsf{P}$, $\pi_1 \in \mathsf{traces}(\mathcal{I}(o_1, p_1))$, and $\pi_2 \in \mathsf{traces}(\mathcal{I}(o_2, p_2))$ such that the following hold for $\sigma_1 = \pi_1|_{\mathsf{M}}$ and $\sigma_2 = \pi_2|_{\mathsf{M}}$:*

(a) *if $M = \mathrm{TSO}$, then either σ_1 or σ_2 has a fence or a RMW event; and*

(b) *if $M = \mathrm{RA}$, then neither σ_1 nor σ_2 is RW, and one of the following holds: (i) either σ_1 or σ_2 has a RMW event; (ii) either σ_1 or σ_2 is not LTF (i.e., has a fence in the middle); (iii) σ_1 is LF and σ_2 is TF; or (iv) σ_1 is TF and σ_2 is LF.*

Since `read` is one-sided non-commutative w.r.t. `write` in both $\mathsf{spec}(\mathsf{Reg})$ and $\mathsf{spec}(\mathsf{MaxReg})$, their respective implementations under TSO and RA are subject to the constraints given in Thm. 5.5. The same holds for the implementations of the `read` and `inc` operations of MC.

To establish the tightness of these lower bounds, we present linearizable wait-free implementations of Reg and MaxReg that are optimal w.r.t. the above bounds: for TSO, it uses only reads, writes, and a single fence at the end of `write`; and for RA, it uses only reads, writes, and a pair of fences at both the beginning and the end of both `write` and `read`.

A Reg object is trivial to implement under SCM and there are MaxReg implementations under SCM [3] with every operation being RBW. We use these implementations as a basis for implementations under TSO and RA as follows:

TSO. For TSO, we utilize a *fence-insertion strategy*, which derives a linearizable TSO implementation of an object from its SCM counterpart by inserting a fence in-between every consecutive pair of write and read, as well as between a final write of an operation (if it exists) and the operation's response. We give full details, prove correctness, and present more examples of applications of this transformation in [14]. Using this strategy, we obtain a TSO implementation of Reg as follows: `write` first writes to a memory location, and then executes a fence, and `read` reads the same memory location and returns the value read. Likewise, to implement MaxReg under TSO, we add a fence at the end of the `write` implementations of [3], and leave their `read` implementation as is.

RA. We augment the TSO implementations above by adding another fence at the beginning of `write` as well as fences at the beginning and the end of `read`. The pseudocode of the MaxReg algorithm appears in §A and its correctness proof can be found in [14]. Further details of the register implementation and its correctness proof appear in [14]. For conciseness, our MaxReg implementation under RA is derived from a simplified version of the algorithm in [3] (with linear step complexity instead of logarithmic as in [3]).

5.2 Two-Sided Non-Commutative Operations and Mutual Exclusion

We next explore implementability of objects with non-weakly mergeable histories. We apply our framework to generalize the "laws of order" (LOO) results of [6]. The next notion of two-sided non-commutativity is a strengthening of one-sided non-commutativity defined above, and is identical to the notion of strong non-commutativity in [6]:

▶ **Definition 5.6.** Two operations $o_1, o_2 \in \mathsf{ops}(O)$ are *two-sided non-commutative in* $\mathsf{spec}(O)$ if there exist history $h_0 \in \mathsf{ComSeqH}(O)$, processes $p_1 \neq p_2$, and response values $u_1 \neq v_1$ and $u_2 \neq v_2$ in $\mathsf{rets}(O)$ such that: (i) $h_0 \cdot (p_1\colon |\underline{o_1 u_1}|\,) \cdot (p_2\colon |\underline{o_2 v_2}|\,) \in \mathsf{spec}(O)$; and (ii) $h_0 \cdot (p_2\colon |\underline{o_2 u_2}|\,) \cdot (p_1\colon |\underline{o_1 v_1}|\,) \in \mathsf{spec}(O)$.

▶ **Example 5.7.** Revisiting Example 1, in a standard set object Set the operations $\mathtt{remove}(v)$ and $\mathtt{remove}(v)$ (for any v) are strongly non-commutative. Indeed, we can take any $p_1 \neq p_2$, $h_0 = (p\colon |\underline{\mathsf{add}(v)}|\,)$ (with any $p \in \mathsf{P}$), $u_1 = u_2 = \mathit{true}$, and $v_1 = v_2 = \mathit{false}$, and we have $(p\colon |\underline{\mathsf{add}(v)}|\,) \cdot (p_1\colon |\underline{\mathtt{remove}(v)\quad \mathit{true}}|\,) \cdot (p_2\colon |\underline{\mathtt{remove}(v)\quad \mathit{false}}|\,) \in \mathsf{spec}(\mathsf{Set})$ and $(p\colon |\underline{\mathsf{add}(v)}|\,) \cdot (p_2\colon |\underline{\mathtt{remove}(v)\quad \mathit{true}}|\,) \cdot (p_1\colon |\underline{\mathtt{remove}(v)\quad \mathit{false}}|\,) \in \mathsf{spec}(\mathsf{Set})$.

▶ **Example 5.8.** Consider a consensus object $\mathsf{Consensus}$ with operations $\mathtt{propose}(0)$ and $\mathtt{propose}(1)$ and return values $\{0,1\}$. Its specification $\mathsf{spec}(\mathsf{Consensus})$ consists of all histories $h \in \mathsf{ComSeqH}(\mathsf{Consensus})$ such that every $\mathtt{propose}(v)$ invoked in h returns the same value, which is either v or the argument of one of the previously invoked $\mathtt{propose}$ operations. The operations $\mathtt{propose}(0)$ and $\mathtt{propose}(1)$ are two-sided non-commutative. Indeed, for any $p_1 \neq p_2$ and $h_0 = \varepsilon$, we have $(p_1\colon |\underline{\mathtt{propose}(0)\quad 0}|\,) \cdot (p_2\colon |\underline{\mathtt{propose}(1)\quad 0}|\,) \in \mathsf{spec}(\mathsf{Consensus})$ and $(p_2\colon |\underline{\mathtt{propose}(1)\quad 1}|\,) \cdot (p_1\colon |\underline{\mathtt{propose}(0)\quad 1}|\,) \in \mathsf{spec}(\mathsf{Consensus})$.

Examples for other objects with consensus number > 1, such as swap, compare-and-swap, fetch-and-add, queues, stacks, are constructed similarly. In [14] we show that deterministic objects with a pair of two-sided non-commutative operations must have consensus numbers > 1. (We conjecture that the converse also holds.)

We prove in [14] that two-sided non-commutative operations imply non-weakly mergeability:

▶ **Lemma 5.9.** *Let O be a deterministic object and $o_1, o_2 \in \mathsf{ops}(O)$ be two-sided non-commutative operations in $\mathsf{spec}(O)$. Then, there exist $h_0 \in \mathsf{ComSeqH}(O)$, processes $p_1 \neq p_2$ and response values $u_1, u_2 \in \mathsf{rets}(O)$ such that $(p_1\colon |\underline{o_1 u_1}|\,)$ and $(p_2\colon |\underline{o_2 u_2}|\,)$ are not weakly mergeable in $\mathsf{spec}(O)$ after h_0.*

We now apply the merge theorem and the properties $\mathrm{SCM}^{\mathsf{w}}, \mathrm{TSO}^{\mathsf{w}}, \mathrm{RA}^{\mathsf{w}}$ from Table 1 to obtain the lower bounds of LOO under SCM along with impossibilities for TSO and RA (see [14] for the proof):

▶ **Theorem 5.10.** *Let O be a deterministic object with a pair of strongly non-commutative operations $o_1, o_2 \in \mathsf{ops}(O)$ in $\mathsf{spec}(O)$. Let $\mathcal{I}$ be a spec-available implementation of O that is consistent under a memory model M. Then, there exist $p_1 \in \mathsf{P}$ and $\pi_1 \in \mathsf{traces}(\mathcal{I}(o_1, p_1))$ such that the following hold for $\sigma_1 = \pi_1|_{\mathsf{M}}$:*

(a) *if $M = \mathrm{SCM}$, then σ_1 either has an RMW or is not RBW; and*

(b) *if $M \in \{\mathrm{TSO}, \mathrm{RA}\}$, then σ_1 either has an RMW or is not LTF (i.e., has a fence in the middle).*

Since deterministic objects with a pair of two-sided non-commutative operations have consensus numbers > 1, their wait-free implementations must rely on RMWs [20]. We therefore consider their obstruction-free implementations to obtain upper bounds in the absence of RMWs.[2] In [14], we show that every object in this class has an obstruction-free implementation under TSO with a fence pattern optimal w.r.t. our lower bounds in

[2] It is known that every deterministic object has read/write obstruction-free linearizable implementations in SCM [21].

contention-free executions, i.e., when a process runs solo for long enough to complete its operation. For that, we use a variant of a universal construction from [33] instantiated on top of a TSO-based obstruction-free consensus algorithm. The latter is obtained from shared memory Paxos [18] using our fence-insertion strategy.

Finally, in §B, we derive lower bounds for mutual exclusion. We define an object Lock that can be implemented by means of an entry section of a mutual exclusion algorithm. We show that Lock has a pair of non-weakly mergeable histories, and apply the merge theorem to obtain the lower bounds of LOO for SCM and their counterparts for TSO and RA. A matching upper bound for TSO is obtained by adding a single fence to the entry section of the Bakery algorithm [28].

5.3 Snapshot and Counter

We next explore implementability of snapshot [1] (Snapshot) and (non-monotone) counter (Counter). The former is known to be universal w.r.t. a large class of objects implementable in read/write SCM [5], and the latter has been studied extensively as a building block for randomized consensus (e.g., [4, 2]).

In §C, we revisit and formalize Example 2, and obtain lower bounds on memory events and fence structure that must be exhibited by any consistent and spec-available implementation of snapshot and counter under the memory models we consider. Specifically, we obtain the following for snapshot:

▶ **Theorem 5.11.** *Let $\mathcal{I}$ be a spec-available implementation of* Snapshot *that is consistent under a memory model M. Then, there exist $p, p' \in \mathsf{P}$, $\pi_1 \in \mathsf{traces}(\mathcal{I}(\mathtt{update}(w), p))$ for some $i \in \{1..m\}$ and $w \in W$, and $\pi_2 \in \mathsf{traces}(\mathcal{I}(\mathtt{scan}, p'))$ such that the following hold for $\sigma_1 = \pi_1|_{\mathsf{M}}$ and $\sigma_2 = \pi_2|_{\mathsf{M}}$:*

(a) *if $M = \mathrm{SCM}$, then $\sigma_1 \cdot \sigma_2$ either has an RMW or is not RBW;*

(b) *if $M = \mathrm{TSO}$, then $\sigma_1 \cdot \sigma_2$ either has an RMW or is not LTF (i.e., has a fence in the middle); and*

(c) *if $M = \mathrm{RA}$, then (i) either σ_1 or σ_2 has an RMW, or (ii) either σ_1 or σ_2 is not LTF.*

We show that a similar lower bounds holds for Counter for $\pi_1 \in \mathsf{traces}(\mathcal{I}(o, p))$ with $o \in \{\mathtt{inc}, \mathtt{dec}\}$ and $\pi_2 \in \mathsf{traces}(\mathcal{I}(\mathtt{read}, p'))$.

The wait-free linearizable implementations of both Snapshot and Counter under SCM are well-known [1, 5]. The implementation of `update` operation uses collect followed by a write, and the implementation of `scan` uses a sequence of three collects. Counter can be implemented on top of a snapshot using a single call to `update` to implement increment and decrement, and a single call to `scan` to implement read. Both implementations exhibit a single read-after-write across a consecutive pair of update and read, and are therefore optimal w.r.t. to the above lower bounds.

To obtain optimal upper bounds for TSO, the above algorithms are modified using the fence-insertion strategy discussed above that inserts a fence at the end of `update` for snapshot, and at the end of the increment and decrement for counter. Optimal implementations under RA are left for future work.

Max-register vs. snapshot and counter. Our analysis yields the first sharp separation between max-register on the one hand and snapshot and counter on the other in terms of their implementability under RA using only reads, writes, and fences. Specifically, as we show above, max-register can be implemented under RA using fences only at the beginning and the end of `read` and `write`. On the other hand, our lower bounds for snapshot and counter show that this fence placement is insufficient to correctly implement these objects under RA. We are unaware of prior results separating these objects. In particular, all of them are equivalent w.r.t. their power to solve consensus under SCM [20].

6 Related Work

Our mergeability approach is inspired by the work of Kawash [25], who showed that, without fences and RMWs, the critical section problem, as well as certain producer/consumer coordination problems, cannot be solved in a variety of weak memory models that were studied at that time (including TSO). However, while Kawash considers specific tasks, we derive a general result by relating mergeability of traces in the underlying memory model to mergeability at the level of the implemented object histories.[3] Moreover, we also use different mergeability properties to differentiate between weak memory models.

We have already discussed how the results of [6], which were based on a covering technique [13], are obtained by a simpler merge-based argument. The main advantage of our approach is its applicability beyond "strongly non-commutative operations" (see Lem. 5.9), as well as the fact that we directly handle weak memory models, which is only implicit in [6]. In addition, [6] is restricted to deterministic objects and implementations, while our merge theorem avoids these assumptions by stating more precise availability requirements.

Through consensus numbers, Herlihy [20] already showed that for some of the objects we consider here, such as sets, queues and stacks, RMW operations are required in any lock-free linearizable implementation. This result does not have any implication for obstruction-freedom. In fact, for every object, there is a read/write obstruction-free linearizable implementation under SCM (as consensus is universal and read/write obstruction-free solvable [20, 21]). Due to our results, if the object has non-weakly mergeable operations, any such implementation cannot be RBW.

For several objects such as snapshot, counter, max register, work stealing and even relaxations of queues, stacks, and data sketches, there have been proposed lock-free or wait-free read/write linearizable (or variants of it) RBW implementations under SCM [5, 1, 3, 15, 16, 34]. None of these works relate the possibility of such implementations with mergeability properties of the objects implemented. Morrison and Afek [30] show how memory fences can be eliminated on TSO in the implementation of work stealing by assuming that the store buffers are bounded in size, and using this bound in the thief implementation to guarantee that a write is propagated to main memory after a number of subsequent writes. In contrast, the store buffers in the TSO model we study are unbounded, and hence their implementation is not considered linearizable.

In the weak memory literature, some works studied *robustness* of concurrent implementations under TSO and RA, where a robust implementation cannot have any non-SCM behaviors [11, 9, 27, 10, 29]. We note, however, that robustness does not entail that linearizability under SCM is transferred to linearizability under TSO or RA (a register implementation that uses one shared variable is robust, but fences are needed to ensure linearizability under TSO and RA). This is different from the fence-insertion strategy in §5.1 that transfers linearizability under SCM to linearizability under TSO. Other works studied alternatives to linearizability for TSO and RA [12, 35, 32], whereas we take standard linearizability as a correctness criterion.

[3] We also note that Kawash's merge strategy for TSO traces is unnecessarily complex, while our proofs directly exploit the local store buffers for avoiding inter-thread communication.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.

2 James Aspnes. Time-and space-efficient randomized consensus. In *PODC*, pages 325–331, New York, NY, USA, 1990. ACM. doi:10.1145/93385.93433.

3 James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, 2012. doi:10.1145/2108242.2108244.

4 James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, 1990. doi:10.1016/0196-6774(90)90021-6.

5 James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *SPAA*, pages 340–349. ACM, 1990. doi:10.1145/97444.97701.

6 Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, pages 487–498. ACM, 2011. doi:10.1145/1926385.1926442.

7 Mirza Ahad Baig, Danny Hendler, Alessia Milani, and Corentin Travers. Long-lived counters with polylogarithmic amortized step complexity. *Distributed Comput.*, 36(1):29–43, 2023. doi:10.1007/S00446-022-00439-5.

8 Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, New York, NY, USA, 2011. ACM. doi:10.1145/1926385.1926394.

9 Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *ESOP*, volume 7792 of *LNCS*, pages 533–553. Springer, 2013. doi:10.1007/978-3-642-37036-6_29.

10 Ahmed Bouajjani, Constantin Enea, Suha Orhun Mutluergil, and Serdar Tasiran. Reasoning about TSO programs using reduction and abstraction. In *CAV*, pages 336–353, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-96142-2_21.

11 Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. Deciding robustness against total store ordering. In *ICALP (2)*, pages 428–440, 2011. doi:10.1007/978-3-642-22012-8_34.

12 Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Concurrent library correctness on the TSO memory model. In *ESOP*, pages 87–107, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-28869-2_5.

13 James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, 1993. doi:10.1006/INCO.1993.1065.

14 Armando Castañeda, Gregory Chockler, Brijesh Dongol, and Ori Lahav. What cannot be implemented on weak memory? *CoRR*, abs/2405.16611, 2024. doi:10.48550/arXiv.2405.16611.

15 Armando Castañeda and Miguel Piña. Read/write fence-free work-stealing with multiplicity. *J. Parallel Distributed Comput.*, 186:104816, 2024. doi:10.1016/J.JPDC.2023.104816.

16 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Set-linearizable implementations from read/write operations: Sets, fetch &increment, stacks and queues with multiplicity. *Distributed Comput.*, 36(2):89–106, 2023. doi:10.1007/s00446-022-00440-y.

17 Edsger W. Dijkstra. EWD123: Cooperating Sequential Processes. Technical report, University of Texas, 1965. URL: http://www.cs.utexas.edu/~EWD/transcriptions/EWD01xx/EWD123.html.

18 Eli Gafni and Leslie Lamport. Disk paxos. *Distrib. Comput.*, 16(1):1–20, February 2003. doi:10.1007/S00446-002-0070-8.

19 Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. doi:10.1145/564585.564601.

20 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

21 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003. doi:10.1109/ICDCS.2003.1203503.

22 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

23 Intel. Intel® 64 and IA-32 architectures software developer's manual. *Volume 3B: system programming guide, Part*, 2(11):1–64, 2011.

24 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL*, pages 175–189, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009850.

25 J. Y. Kawash. *Limitation and capabilities of weak memory consistency systems*. PhD thesis, University of Calgary, 2000. doi:10.11575/PRISM/19939.

26 Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In *POPL*, pages 649–662, New York, NY, USA, 2016. ACM. doi:10.1145/2837614.2837643.

27 Ori Lahav and Roy Margalit. Robustness against release/acquire semantics. In *PLDI*, pages 126–141, New York, NY, USA, 2019. ACM. doi:10.1145/3314221.3314604.

28 Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974. doi:10.1145/361082.361093.

29 Roy Margalit and Ori Lahav. Verifying observational robustness against a C11-style memory model. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434285.

30 Adam Morrison and Yehuda Afek. Fence-free work stealing on bounded TSO processors. In *ASPLOS*, pages 413–426. ACM, 2014. doi:10.1145/2541940.2541987.

31 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, pages 391–407, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-03359-9_27.

32 Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290381.

33 Michel Raynal. Distributed universal construcitons: a guided tour. *Bulleting of EATCS: Distributed Computing Column*, (121), 2011.

34 Arik Rinberg and Idit Keidar. Intermediate value linearizability: A quantitative correctness criterion. In *DISC*, volume 179 of *LIPIcs*, pages 2:1–2:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.2.

35 Abhishek Kr Singh and Ori Lahav. An operational approach to library abstraction under relaxed memory concurrency. *Proc. ACM Program. Lang.*, 7(POPL):1542–1572, 2023. doi:10.1145/3571246.

36 SPARC International Inc. *The SPARC architecture manual (version 9)*. Prentice-Hall, 1994.

A Fence-optimal Max Register Under RA

The pseudocode of a linearizable wait-free implementation of MaxReg under RA is given in Algorithm 1. The function *collect*(M) reads one by one, in an arbitrary order, the entries of M, and returns an array with the read values. The algorithm is fence-optimal. It uses one fence at the beginning and one fence at the end of every operation, thus matching the lower bounds of Thm. 5.5. The correctness proof appears in [14].

B Mutual Exclusion

We use the merge theorem (Thm. 4.8) for the case of non-weakly mergeable histories and the mergeability results for the memory models to establish minimum synchronization

Algorithm 1 MaxReg implementation in RA. Algorithm for proces p_i.

```
Shared variables:
    $int[n]\ \ M = [0, \dots, 0]$

procedure READ( )
    fence()
    $m[] = collect(M)$
    fence()
    return $max(m[])$

procedure WRITE($v$)
    fence()
    $m[] = collect(M)$
    if $max(m[]) < v$ then
        $M[i] = v$
    fence()
    return ack
```

requirements for mutual exclusion. Our result for SCM reproves the corresponding lower bound of [6].

Consider a (non-standard) lock object Lock with ops(Lock) $\triangleq$ {acquire} and rets(Lock) $\triangleq$ {ack}. Its specification is given by

$$\mathsf{spec}(\mathsf{Lock}) \triangleq \{\varepsilon\} \cup \{(p\colon \left|\underline{\texttt{acquire}\quad}\right|) \mid p \in \mathsf{P}\}.$$

The histories $(p\colon \left|\underline{\texttt{acquire}\quad}\right|)$ and $(p'\colon \left|\underline{\texttt{acquire}\quad}\right|)$ where $p \neq p'$ are not weakly mergeable. Thus, by the merge theorem and properties SCM$^{\mathsf{w}}$, TSO$^{\mathsf{w}}$, and RA$^{\mathsf{w}}$ in Table 1, we have:

▶ **Theorem B.1.** *Let $\mathcal{I}$ be a spec-available implementation of* Lock *that is consistent under a memory model M. Then, there exist $p \in \mathsf{P}$ and $\pi \in \mathsf{traces}(\mathcal{I}(\texttt{acquire}, p))$ such that the following hold for $\sigma = \pi|_{\mathsf{M}}$:*

(a) *if $M = \mathrm{SCM}$, then σ either has an RMW event or is not RBW; and*

(b) *if $M \in \{TSO, \mathrm{RA}\}$, then σ either has an RMW event or a fence.*

The proof of this theorem is identical to that of Thm. 5.10, which appears in [14]. Since the implementation of the entry section of a mutual exclusion algorithm can be used to implement acquire, we obtain that entry section of a solo-terminating mutual exclusion algorithm on SCM has to use a RAW pattern or an RMW; and on TSO or RA, it must use an RMW or a fence.

There exist many algorithms implementing starvation-free mutual exclusion under SCM, which use the RAW pattern to implement the entry section. As before, their counterparts under TSO can be obtained by adding a fence between every pair of consecutive write and read (§5.1). For example, the transformation of Bakery algorithm [28] only requires a single fence to separate a write-only block at the beginning of the entry section from the read-only block right afterwards. The resulting implementation is therefore tight. Mutual exclusion under RA with an RMW or a fence has several verified implementations [27].

C Lower and Upper Bounds for Snapshot and Counter

Lower bounds for snapshot. Consider a (single-writer) snapshot object Snapshot storing a vector of a length $|\mathsf{P}|$ over a set of values W (also represented as function in $\mathsf{P} \to W$) with the initial vector of $\langle \bot, \dots, \bot \rangle$. The operations are $\{\texttt{update}(w) \mid w \in V\} \cup \{\texttt{scan}\}$, and its return values are $\{\texttt{ack}\} \cup (\mathsf{P} \to W)$. The specification spec(Snapshot) consists of all complete sequential histories where each scan event returns v such that $v(p)$ is the value written by the last preceding update by process p, or $\bot$ if no such update exists.

▶ **Proposition C.1.** *Let $w, w' \in W$, $p_1, p_2, p_3 \in \mathsf{P}$, and $h_1, h_2 \in \mathsf{ComH}(\mathsf{Snapshot})$, such that $w \neq w'$, $i \neq j$, $\mathsf{proc\text{-}set}(h_1) \cap \mathsf{proc\text{-}set}(h_2) = \emptyset$, and the following hold:*

- *$h_1 \sqsubseteq (p_1\colon \vdash\overset{\texttt{update}(w)}{\rule{5em}{0.4pt}}\dashv) \cdot (p_3\colon \vdash\overset{\texttt{scan}\ \ v}{\rule{2.5em}{0.4pt}}\dashv)$, where $v = \lambda p.$ **if** $p = p_1$ **then** w **else** $\bot$; and*
- *$h_2 \sqsubseteq (p_2\colon \vdash\overset{\texttt{update}(w')}{\rule{5em}{0.4pt}}\dashv) \cdot (p_2\colon \vdash\overset{\texttt{scan}\ \ v'}{\rule{2.5em}{0.4pt}}\dashv)$, where $v' = \lambda p.$ **if** $p = p_2$ **then** w' **else** $\bot$.*

Then, h_1 and h_2 are not weakly mergeable in $\mathsf{spec}(\mathsf{Snapshot})$ after ε.

Next, we use the merge theorem (instantiated for the case of non-weakly mergeable histories) together with Prop. C.1 and the mergeability results SCM^{w}, TSO^{w}, and RA^{w} from Table 1 to establish lower bounds on implementability of snapshot.

▶ **Theorem 5.11.** *Let $\mathcal{I}$ be a spec-available implementation of $\mathsf{Snapshot}$ that is consistent under a memory model M. Then, there exist $p, p' \in \mathsf{P}$, $\pi_1 \in \mathsf{traces}(\mathcal{I}(\texttt{update}(w), p))$ for some $i \in \{1..m\}$ and $w \in W$, and $\pi_2 \in \mathsf{traces}(\mathcal{I}(\texttt{scan}, p'))$ such that the following hold for $\sigma_1 = \pi_1|_{\mathsf{M}}$ and $\sigma_2 = \pi_2|_{\mathsf{M}}$:*

(a) *if $M = \text{SCM}$, then $\sigma_1 \cdot \sigma_2$ either has an RMW or is not RBW;*

(b) *if $M = \text{TSO}$, then $\sigma_1 \cdot \sigma_2$ either has an RMW or is not LTF (i.e., has a fence in the middle); and*

(c) *if $M = \text{RA}$, then (i) either σ_1 or σ_2 has an RMW, or (ii) either σ_1 or σ_2 is not LTF.*

Proof. First, consider the case of $M \in \{\text{SCM}, \text{TSO}\}$. Let p_1, p_2 be distinct processes and consider the histories

$$h_1 = (p_1\colon \vdash\overset{\texttt{update(1)}}{\rule{5em}{0.4pt}}\dashv) \cdot (p_1\colon \vdash\overset{\texttt{scan}\ \ v}{\rule{2.5em}{0.4pt}}\dashv) \quad \text{and} \quad h_2 = (p_2\colon \vdash\overset{\texttt{update(1)}}{\rule{5em}{0.4pt}}\dashv) \cdot (p_2\colon \vdash\overset{\texttt{scan}\ \ v'}{\rule{2.5em}{0.4pt}}\dashv),$$

where $v = \lambda p.\,\textbf{if}\ p = p_1\ \textbf{then}\ w\ \textbf{else}\ \bot$ and $v' = \lambda p.\,\textbf{if}\ p = p_2\ \textbf{then}\ w'\ \textbf{else}\ \bot$. Then, by Prop. C.1, h_1 and h_2 are not weakly mergeable in $\mathsf{spec}(\mathsf{Snapshot})$ after $h_0 = \varepsilon$. Clearly, we also have $h_1, h_2 \in \mathsf{spec}(\mathsf{Snapshot})$, and since $\mathcal{I}$ is spec-available, it is available w.r.t. both h_1 and h_2.

Thus, by Thm. 4.8, there exist $\pi_1', \pi_2' \in \mathsf{traces}(\mathcal{I})$ such that $h_1 = \pi_1'|_{\mathsf{Snapshot}}$ and $h_2 = \pi_2'|_{\mathsf{Snapshot}}$, and $\sigma_1' = \pi_1'|_{\mathsf{M}}$ and $\sigma_2' = \pi_2'|_{\mathsf{M}}$ are not weakly mergeable in M. Observe that $\pi_1' = \pi_1 \cdot \pi_2$ where $\pi_1 \in \mathsf{traces}(\mathcal{I}(\texttt{update}(1), p_1))$ and $\pi_2 \in \mathsf{traces}(\mathcal{I}(\texttt{scan}, p_1))$. Let $\sigma_1 = \pi_1|_{\mathsf{M}}$ and $\sigma_2 = \pi_2|_{\mathsf{M}}$. Then, $\sigma_1' = \sigma_1 \cdot \sigma_2$. Thus, the required follows properties SCM^{w} and TSO^{w} in Table 1.

Next, we consider the case of $M = \text{RA}$. Let $p_1, p_2, p_3 \in \mathsf{P}$ be distinct processes, and consider the histories:

$h_1 = \langle p_1{:}\texttt{inv}(\texttt{update}(1)), p_3{:}\texttt{inv}(\texttt{scan}), p_1{:}\texttt{res}(\texttt{ack}), p_3{:}\texttt{res}(v)\rangle$ and
$h_2 = (p_2\colon \vdash\overset{\texttt{update(2)}}{\rule{5em}{0.4pt}}\dashv) \cdot (p_2\colon \vdash\overset{\texttt{scan}\ \ v'}{\rule{2.5em}{0.4pt}}\dashv)$,
where $v = \lambda p.\,\textbf{if}\ p = p_1\ \textbf{then}\ w\ \textbf{else}\ \bot$ and $v' = \lambda p.\,\textbf{if}\ p = p_2\ \textbf{then}\ w'\ \textbf{else}\ \bot$. Then, by Prop. C.1, h_1 and h_2 are not weakly mergeable in $\mathsf{spec}(\mathsf{Snapshot})$ after $h_0 = \varepsilon$. Note that $h_2 \in \mathsf{spec}(\mathsf{Snapshot})$. Consider the following sequential history of $\mathsf{spec}(\mathsf{Snapshot})$:

$$h^1_{\mathsf{seq}} = (p_1\colon \vdash\overset{\texttt{update(1)}}{\rule{5em}{0.4pt}}\dashv) \cdot (p_3\colon \vdash\overset{\texttt{scan}\ \ v}{\rule{2.5em}{0.4pt}}\dashv)$$

By assumption, $\mathcal{I}$ is available w.r.t. h_2 and h^1_{seq}.

Then, by Thm. 4.8, there exist π_1 and π_2 such that:

- $\pi_i|_{\mathsf{Snapshot}} = h_i$ for $i \in \{1, 2\}$.
- $\pi_i \in \mathsf{traces}(\mathcal{I})$ for $i \in \{1, 2\}$.
- $\pi_i|_{\mathsf{M}} \in \mathsf{traces}(\text{SCM})$ for $i \in \{1, 2\}$.
- $\mathsf{proc\text{-}set}(\pi_i) = \mathsf{proc\text{-}set}(h_i)$ for $i \in \{1, 2\}$.

- For every $\pi_1' \in \mathsf{reorder}_{\mathsf{sproc}}(\pi_1)$ such that $\pi_1'|_{\mathsf{M}} \in \mathsf{traces}(\mathrm{SCM})$ and $\pi_1|_{\mathsf{Snapshot}} = h_1$ and $\pi_2' \in \mathsf{reorder}_{\mathsf{sproc}}(\pi_2)$ such that $\pi_2'|_{\mathsf{M}} \in \mathsf{traces}(\mathrm{SCM})$ and $\pi_2|_{\mathsf{Snapshot}} = h_2$, $\pi_1'|_{\mathsf{M}}$ and $\pi_2'|_{\mathsf{M}}$ are not weakly mergeable in RA.

  Let π_1' be the sequence obtained from π_1 by:
- moving $\langle p_1, \texttt{inv}(\texttt{update}(1))\rangle$, $\langle p_3, \texttt{inv}(\texttt{scan})\rangle$ and all leading fences to the beginning of the sequence; and
- moving $\langle p_1, \texttt{res}(\texttt{ack})\rangle$, $\langle p_3, \texttt{res}(v)\rangle$ and all trailing fences to the end of the sequence.

In this rearrangement we keep the internal order among moved events as it is in π_1. Then, $\pi_1' \in \mathsf{reorder}_{\mathsf{sproc}}(\pi_1)$ and $\pi_1|_{\mathsf{Snapshot}} = h_1$. Moreover, among memory events, we only moved fences which are no-ops under SCM. Thus, $\pi_1|_{\mathsf{M}} \in \mathsf{traces}(\mathrm{SCM})$ implies $\pi_1'|_{\mathsf{M}} \in \mathsf{traces}(\mathrm{SCM})$. By taking $\pi_2' = \pi_2$, we obtain that $\pi_1'|_{\mathsf{M}}$ and $\pi_2'|_{\mathsf{M}}$ are not weakly mergeable in RA. Finally, by property RA^{w} in Table 1, we obtain that $\pi_1'|_{\mathsf{M}}$ is not LTF, or it contains some RMW event. This implies that either $\pi_1'|_{\mathsf{M}}|_{p_1}$ or $\pi_1'|_{\mathsf{M}}|_{p_3}$ are not LTF or contain some RMW event. ◀

Lower bounds for counter. Consider a counter object $\mathsf{Counter}$ with the initial value of 0, and the increment (`inc`), decrement (`dec`), and read (`read`) operations. Then, we have:

▶ **Proposition C.2.** *Let $p_1, p_2, p_3 \in \mathsf{P}$ and $h_1, h_2 \in \mathsf{ComH}(\mathsf{Counter})$ such that $\mathsf{proc\text{-}set}(h_1) \cap \mathsf{proc\text{-}set}(h_2) = \emptyset$ and the following hold:*

- $h_1 \sqsubseteq (p_1\colon |\underline{\texttt{inc}\qquad\quad}|) \cdot (p_3\colon |\underline{\texttt{read}\qquad 1}|)$*; and*
- $h_2 \sqsubseteq (p_2\colon |\underline{\texttt{dec}\qquad\quad}|) \cdot (p_2\colon |\underline{\texttt{read}\quad -1}|)$

Then, h_1 and h_2 are not weakly mergeable in $\mathsf{spec}(\mathsf{Counter})$ after ε.

Then, the following can be obtained by instantiating the proof of Thm. 5.11 to use Prop. C.2.

▶ **Theorem C.3.** *Let $\mathcal{I}$ be a spec-available implementation of $\mathsf{Counter}$ that is consistent under a memory model M. Then, there exist $p, p' \in \mathsf{P}$, $\pi_1 \in \mathsf{traces}(\mathcal{I}(o, p))$ where $o \in \{\texttt{inc}, \texttt{dec}\}$ and $\pi_2 \in \mathsf{traces}(\mathcal{I}(\texttt{read}, p'))$ such that the following hold for $\sigma_1 = \pi_1|_{\mathsf{M}}$ and $\sigma_2 = \pi_2|_{\mathsf{M}}\}$:*

(a) *if $M = \mathrm{SCM}$, then $\sigma_1 \cdot \sigma_2$ either has a RMW event or is not RBW; and*

(b) *if $M = \mathrm{TSO}$, then $\sigma_1 \cdot \sigma_2$ has either a RMW event or is non-LTF (i.e., has a fence in the middle).*

(c) *if $M = \mathrm{RA}$, then (i) either σ_1 or σ_2 has an RMW, or (ii) either σ_1 or σ_2 is non-LTF.*

Upper bounds. There is a wait-free snapshot implementation [1] that is linearizable under SCM, in which `scan` performs a sequence of reads, and `update` performs a sequence of reads followed by a write, Using the fence insertion strategy in §5.1, a linearizable wait-free implementation of snapshot under TSO is obtained from such implementations by adding a single fence at the end of `update`.

▶ **Theorem C.4.** *For $M \in \{\mathrm{SCM}, \mathrm{TSO}\}$, there exists a linearizable wait-free implementation of snapshot $\mathsf{Snapshot}_M$ under M such that:*

(a) *$\mathsf{Snapshot}_{\mathrm{SCM}}$ uses only a sequence of reads followed by a write to implement `update` and only reads to implement `scan`, and*

(b) *$\mathsf{Snapshot}_{\mathrm{TSO}}$ uses only a sequence of reads followed by a write and a fence at the end to implement `update`, and only reads to implement `scan`.*

Observe that Thm. C.4 (a) implies that any pair of consecutive `update` and `scan` is RBW, which is tight in the lower bound of Thm. 5.11 (a). Likewise, Thm. C.4 (b) is tight in the lower bound of Thm. 5.11 (b), which stipulates that a fence is needed somewhere within consecutively executed `update` and `scan`.

A linearizable wait-free counter can be implemented on top of a snapshot instance as follows: each process p_i stores its contribution to the current counter value in a local variable c_i initialized to 0. To increment (resp., decrement) the counter, p_i increments (resp., decrements) c_i, and then invokes `update`(c_i) to share its contribution with other processes. To read the counter, a process calls `scan` and returns the sum of the values stored in the returned vector.

▶ **Theorem C.5.** *For $M \in \{\mathrm{SCM}, \mathrm{TSO}\}$, there exists a linearizable wait-free implementation of counter* $\mathsf{Counter}_M$ *under M such that:*

(a) $\mathsf{Counter}_{\mathrm{SCM}}$ *uses only writes to implement* `inc` *and* `dec` *and only reads to implement* `read`*, and*

(b) $\mathsf{Counter}_{\mathrm{TSO}}$ *uses only writes and a fence at the end to implement* `inc` *and* `dec`*, and only reads to implement* `read`*.*

As in the case of snapshot, the synchronization strategy stipulated by this result is optimal w.r.t. the lower bound of Thm. C.3. The optimal implementations of snapshot and counter under RA are left for future work.

Faster Cycle Detection in the Congested Clique

Keren Censor-Hillel
Department of Computer Science, Technion, Haifa, Israel

Tomer Even
Department of Computer Science, Technion, Haifa, Israel

Virginia Vassilevska Williams
Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

We provide a fast distributed algorithm for detecting h-cycles in the Congested Clique model, whose running time decreases as the number of h-cycles in the graph increases. In undirected graphs, constant-round algorithms are known for cycles of even length. Our algorithm greatly improves upon the state of the art for odd values of h. Moreover, our running time applies also to directed graphs, in which case the improvement is for all values of h. Further, our techniques allow us to obtain a triangle detection algorithm in the quantum variant of this model, which is faster than prior work.

A key technical contribution we develop to obtain our fast cycle detection algorithm is a new algorithm for computing the product of many pairs of small matrices in parallel, which may be of independent interest.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Distributed algorithms

Keywords and phrases triangle detection, cycle detection, distributed computing, Congested Clique, quantum computing, Fast matrix multiplication, Fast rectangular matrix multiplication

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.12

Related Version *Full Version*: https://www.arxiv.org/abs/2408.15132

Funding *Keren Censor-Hillel*: The research is supported in part by the Israel Science Foundation (grant 529/23).
Virginia Vassilevska Williams: Supported by NSF Grant CCF-2330048, BSF Grant 2020356, and a Simons Investigator Award.

Acknowledgements We would like to thank the anonymous reviewers for their feedback.

1 Introduction

Finding small subgraph patterns is a fundamental computational task, with a multitude of applications for uncovering connections between elements in a data set. Research has been thriving, addressing the complexity of different variants of subgraph isomorphism for fixed size subgraph patterns H in a larger host graph G: detecting whether a copy of H exists, listing all of its copies, counting the number of occurrences, and more.

In this paper, we provide a fast distributed algorithm for detecting h-cycles in the Congested Clique model [37], in which n machines communicate by sending $\mathcal{O}(\log n)$-bit messages to each other, in synchronous rounds.

The pioneering work of [19] showed that all copies of any fixed h-vertex graph H in an n node graph can be listed in this model within $\mathcal{O}(n^{1-2/h})$ rounds. This result of course applies also to the detection variant. For the case when H is a cycle, [12] provided an h-cycle detection algorithm running in $2^{\mathcal{O}(h)}n^{\rho}$ rounds, for both undirected and directed graphs (henceforth digraphs). Here, ρ is the exponent of distributed fast matrix multiplication (FMM) in the Congested Clique model, i.e., the value such that $\mathcal{O}(n^{\rho})$ rounds are sufficient for multiplying two $n \times n$ matrices. The value of ρ is currently known to be at most $1 - 2/\omega$ where ω is the centralized fast matrix multiplication exponent, and since $\omega \leq 2.371552$ [2],

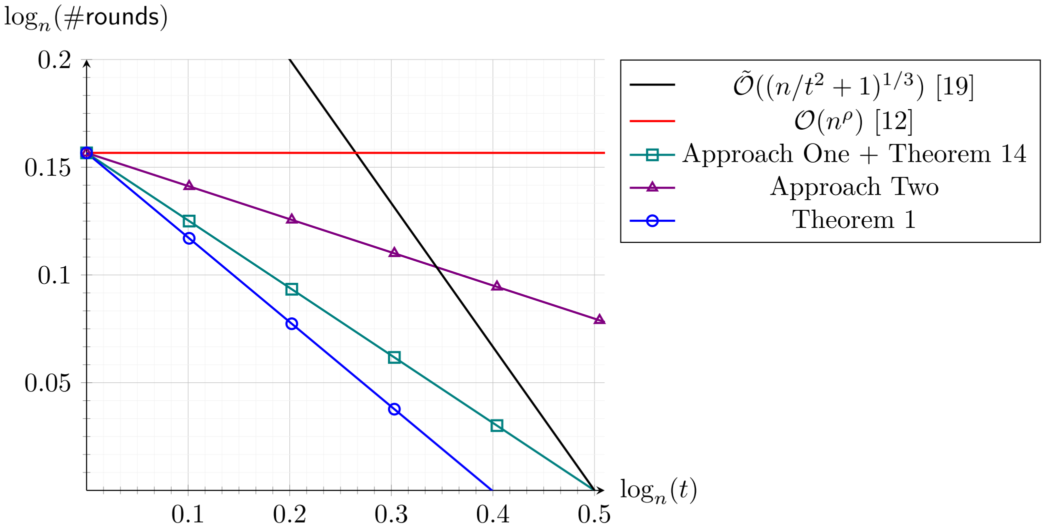


Figure 1 An illustrative comparison between our results and prior work, for the case of triangles. For each algorithm, we plot the base-n logarithm of the number of rounds as a function of the base-n logarithm of the number of triangles.

we get a bound for ρ of 0.15667. In the case of 4-cycles in undirected graphs, [12] obtained a constant-round detection algorithm, and this result was later generalized by [10] to hold for detection of any even-length cycle in undirected graphs.

This leaves the complexity of odd-cycle detection as an open question, as well as the detection of cycles of any length in digraphs. For triangles, [19] showed a detection algorithm that completes within $\tilde{\mathcal{O}}(n^{1/3}/(t^{2/3}+1))$ rounds, w.h.p.[1], where t is the number of triangles. Since [20] hints that lower bounds for H detection in this model are not within reach, it remains open whether the above is optimal.

Question: *For a given graph H, is there a faster H-detection algorithm when the number of instances of H in the input graph is large?*

We answer this question in the affirmative, providing a fast h-cycle detection algorithm whose complexity decreases as the number t of instances of H grows. Our algorithm has the same running time for detecting h-cycles in graphs as well as in digraphs. For triangles, the complexity of our algorithm greatly improves upon that of [19]. For larger odd cycles in graphs, as well as cycles of any length in digraphs, to the best of our knowledge, this is the first improvement over [12].

An important insight of our main technical contribution is to identify a new refined parameter as a key player for detection: the number of vertices x that participate in an h-cycle. Below, we elaborate on our result and technical approach.

1.1 Our Contributions and Technical Overview

To frame our technical contributions, we first briefly overview the two previous approaches for the case of *triangles*. In [19], an $\tilde{\mathcal{O}}(n^{1/3}/(t^{2/3}+1))$-round algorithm is presented, where t is the number of triangles. The algorithm samples n induced subgraphs, and for each sample

[1] High probability in this paper refers to a probability that is at least $1-1/n^c$ for some constant $c \geq 1$.

it checks for a triangle by letting a dedicated vertex collect the edges of the sample. In [12], a $2^{\mathcal{O}(h)}n^{\rho}$-round algorithm is presented which employs fast matrix multiplication over the entire graph.

Warm-up. As a warm-up, consider the following combination of these approaches to get the best of both worlds, leading to an algorithm that completes in $\tilde{\mathcal{O}}(n^{\rho}/(t^{2\rho}+1))$ rounds, which is already an improvement over the prior state of the art (recall that $\rho < 1/3$). To obtain this, we sample only t^2 induced subgraphs to which each vertex is added with probability $1/t$. Using the second moment method, we can show that at least one sampled subgraph contains a triangle with probability $\Omega(1)$. To check if each subgraph contains a triangle we use matrix multiplication, and so no vertex has to collect the edges of an entire sample. To compute the product of t^2 square matrices of size n/t, we develop a new algorithm, which computes the product of s pairs of square matrices of size k in $\mathcal{O}(n^{\rho-2}\cdot k^2\cdot s^{1-\rho})$ rounds. This algorithm may be of independent interest.

Another natural approach to consider is one that samples a subset of vertices and checks whether any of these vertices participates in a triangle, rather than attempting to sample a complete triangle. Here, we can sample each vertex with probability $1/t^{1/3}$ and check whether it is in a triangle by invoking rectangular matrix multiplication. While this too improves upon the state of the art for some graphs, it is always slower than our first approach. Note that trying to reduce the running time by sampling with a probability that is smaller than $1/t^{1/3}$ would reach a dead-end since it is not likely to hit any vertex in a triangle in case all t triangles are induced by a clique of $t^{1/3}$ vertices. See Figure 1 for a comparison of the two approaches, as well as the previous algorithms, and our new algorithm.

A caveat is that the first approach does not extend for h-cycles, as the number of samples it needs to perform increases with h. For example, for C_4 detection, if we sample uniformly random induced subgraph with n/t vertices, it contains a copy of C_4 with probability at least roughly $1/t^3$ which means we have to sample at least t^3 subgraphs to ensure that we find a copy of C_4 with a constant probability. This is slower than the previous best known algorithm of [12] which takes $2^{\mathcal{O}(h)}n^{\rho}$ rounds. In other words, the first approach is slower as h is larger.

Our contribution. Our key insight is that we can further boost these two approaches such that they complement each other, in the following sense. For a fixed value of t, the first approach is better when the number of vertices that participate in a triangle, which we denote by x, is small, while the second approach is better when x is large. This refinement of considering the parameter x along with t allows us to bring these two approaches a big leap forward by obtaining a faster algorithm for triangle detection, as well as an algorithm for h-cycle detection for longer cycles, in both graphs and digraphs.

Our first algorithm, which we refer to as Find-Cycle, follows the first approach. It samples $s = x^3/t$ subsets of vertices $(U_1,\dots,U_s)$, by adding each vertex to U_i independently with probability $1/x$. The algorithm then checks for every $i \in [s]$ if $G[U_i]$ contains a triangle. This involves computing the product of s pairs of square matrices of size n/x each, which we do in $\tilde{\mathcal{O}}(n^{\rho-2}\cdot(n/x)^2\cdot s^{1-\rho}) = \tilde{\mathcal{O}}(n^{\rho}\cdot(1/x)^2\cdot(x^3/t)^{1-\rho})$ rounds. Using the second moment method (which is very similar to Chebyshev's inequality) we show that at least one of the s induced subgraphs contains a triangle with a constant probability.

Our second algorithm, which we refer to as Find-Vertex-In-Cycle, follows the second approach. It samples a subset of vertices S, by adding each vertex to S independently with probability $1/x$. The algorithm then checks if one of the vertices from S participates in a triangle by computing the product of a rectangular matrix of size $n/x \times n$ and a square matrix of size n. Interestingly, the algorithm Find-Vertex-In-Cycle also achieves the same round complexity, as a function of x, for h-cycle detection, for $h = \mathcal{O}(1)$.

Our final algorithm alternates between the two algorithms until one of them terminates. Among all n vertices with t triangles, the final algorithm is the slowest when the two algorithms have the same round complexity, which happens when $x^{3-1.82408} = \Theta(t)$.

The following states the running time of our fast h-cycle detection algorithm, and is proven in Appendix A.

▶ **Theorem 1** (h-Cycle Detection). *Let G be a (directed) graph with t copies of h-cycles. There is a randomized Congested Clique algorithm for h-cycle detection, which takes $\tilde{\mathcal{O}}(h^{\mathcal{O}(h)} \cdot n^{0.1567}/(t^{\frac{0.4617}{h-1.82408}} + 1))$ rounds w.h.p.*

Here, the constants $0.1567, 0.4617$ arise from the complexity of rectangular multiplication. We are able to show that the product of a rectangular matrix of size $k \times n$ and a square matrix can be computed in $O(n^{0.1567}/k^{0.4617})$ rounds, using the formula of [27] and an adaptation of the code of [5]. To get a flavor of the above complexity, note that a crucial implication of Theorem 1 is that we detect a triangle in $\tilde{\mathcal{O}}(1)$ rounds for graphs with at least $t = \Omega(n^{0.3992})$ triangles, improving upon the previously known threshold of $t = \Omega(n^{1/2})$ from [19].

Many Matrix Multiplications in Parallel. To implement our Find-Cycle algorithm, we need to compute the product of many small random square matrices, which are submatrices of the adjacency matrix of the input graph. We state this informally in the following theorem.

▶ **Theorem 2** (Informal). *Let k, s be two integers such that $k \in [\sqrt{n}, n]$, and $s \leq (n/k)^2$. Then, in the Congested Clique model, the n vertices can compute the product of s pairs of square matrices of size k in $\mathcal{O}(n^{\rho-2} \cdot k^2 \cdot s^{1-\rho})$ rounds, given that the input is distributed among the vertices in a "balanced" manner.*

The formal definitions and the proof of Theorem 2 appear in Section 3, as well as the definitions and claims we need for the proof of Theorem 1.

The conceptual contribution of Theorem 2 is as follows. On one hand, it is known that n vertices can compute the product of n pairs of matrices of size $\sqrt{n}$ in a constant number of rounds, given that the input is balanced, by letting the i-th vertex collect all the entries of the i-th pair of matrices and computing their product. On the other hand, n vertices can also compute the product of one square matrix of size n in $\mathcal{O}(n^{\rho})$ rounds, as shown in [12]. Theorem 2 gives a smooth trade-off between these two extremes.

Note that [27] provides an algorithm for computing the product of s pairs of square matrices of size n in $\mathcal{O}(n^{\rho} \cdot s^{1-\rho})$ rounds for $s \leq n$. The paper also provides an algorithm for computing the product of s pairs of *rectangular* matrices of sizes $n \times m$ and $m \times n$, where the bound on the round complexity is more involved and is not given by an analytic expression, see Section 3.3 for a discussion. Moreover, we need to compute the product of square matrices of size smaller than n, which is not covered by the above algorithm.

Before providing intuition about our proof of Theorem 2, we explain what we mean by a balanced input and how the output should be distributed. Given a set of s pairs of square matrices $\mathcal{Q} = \{(S_i, T_i)\}_{i\in[s]}$ of size k each, where $sk^2 \leq n^2$, we think of the input as a "flat" array of sk^2 entries. The input is distributed as follows. The input given to the first vertex is the first n entries in this array. The second vertex gets the next n entries and so on. For the output $\{(P_i)\}_{i\in[s]}$, where $P_i = S_i \cdot T_i$ for $i \in [s]$, we again transform the set of output matrices into a flat array and let each vertex learn distinct consecutive n entries from it. Note that as long as each vertex holds unique n entries from the input, and every vertex knows which entries from the input every other vertex holds, then the input can be redistributed in $\mathcal{O}(1)$ rounds, using Lemma 8. We therefore define a balanced input as such.

▶ **Definition 3** (Balanced Input). *An input for n vertices is* balanced *if it is partitioned between the vertices such that each vertex holds at most n (unique) entries from the input, and every vertex knows which entries from the input are held by every other vertex.*

Now we can give the intuition behind the proof of Theorem 2. We partition the n vertices into s sets of size n/s each. For every $i \in [s]$ we call the i-th set in the partition the i-th *team*. The i-th team is responsible for computing the product (S_i, T_i) (this partitioning method is similar to that of [27]). After partitioning, the problem boils down to computing one product of square matrices of size k using n/s vertices with bandwidth of size $s \log n$, which we solve by extending the work of [12], which considers only the product of square matrices of size equal to the number of vertices.

Recall that our main motivation for this tool of Theorem 2 is to implement the algorithm Find-Cycle. That is, given a graph G with n vertices, we sample s subsets of vertices $(U_1, \ldots, U_s)$, where each vertex joins each set independently with probability p. Each set U_i defines an induced subgraph $G[U_i]$ with an adjacency matrix A_i. We need to compute $(A_i)^h$ for every $i \in [s]$, and we denote the set $\mathcal{Q} = \{(A_i, A_i)\}_{i\in[s]}$ as the input, where we assume $s \leq 1/p^2$. In order to implement the algorithm Find-Cycle using Theorem 2, we need to show that $\mathcal{Q}$ is a balanced input. However, this does not precisely hold, but we can show a sufficient guarantee of $\mathcal{Q}$ being "almost" balanced w.h.p., in which the requirements in Definition 3 are weakened. These weaker conditions still allow us to quickly redistributed the elements into a balanced input in $\mathcal{O}(\log n)$ rounds w.h.p.

To conclude, we obtain a fast algorithm for h-cycle detection, which beats the previous state-of-the art for odd-length cycles, as well as directed cycles. The algorithm is faster as the number of copies of h-cycles increases, where the key parameter for the algorithm and the analysis is the number of vertices in the graph that participate in an h-cycle.

Triangle Detection in the Quantum Congested Clique Model. Our new matrix multiplication tool turns out to be helpful for additional tasks. In the quantum setting, we obtain the following in the Quantum Congested Clique model, which is similar to the Congested Clique model, but the vertices exchange messages of $\mathcal{O}(\log n)$ qubits in each round instead of standard bits.

▶ **Theorem 4.** *There exists a Quantum Congested Clique $\tilde{\mathcal{O}}((n/(t^2+1))^{3\rho/4})$-round algorithm for triangle detection, with a success probability of at least $1/2$.*

Due to space considerations, the proof of Theorem 4 is deferred to the full version of the paper. Theorem 4 obtains a $\tilde{\mathcal{O}}(n^{3\rho/4})$-round algorithm that remains effective even when $t = 0$. This is faster than the previous state-of-the-art algorithm from [12], which takes $\mathcal{O}(n^{\rho})$ rounds and does not leverage the additional capabilities that the Quantum Congested Clique model offers. For subgraphs other than triangles, there are detection algorithms in the Quantum Congested Clique which use the extra power of the model. For example, [9] provides an algorithm for larger clique detection. Moreover, in the quantum Congest model, there are algorithms for clique and cycle detection [32, 26], which are faster than their Congest counterparts.

Our algorithm uses a Grover search [30], which is a quantum algorithm to find an element in an unsorted list of size L, while accessing only $\sqrt{|L|}$ entries from the list. Generally, given a function $f : X \to \{0, 1\}$ and a universe X, Grover search is a quantum algorithm which finds an $x \in X$ such that $f(x) = 1$ (assuming such x exists), by querying f at most $\sqrt{|X|}$ times w.h.p. In [35] a distributed implementation of Grover search was provided, for the quantum Congest model, which was later extended to the Quantum Congested Clique model in [31, 9].

We provide an overview of our algorithm, which has two steps. In the first step, we sample $s = 1/t^2$ random induced subgraphs $(U_1, \ldots, U_s)$, where each vertex joins every set independently with probability $1/t$. We partition the vertices into s sets, each of n/s vertices, which we call teams. For $i \in [s]$, the i-th team uses Grover search to detect a triangle in the subgraph $G[U_i]$, as follows. It samples $\ell = 8 \log n/q^3$ subsets of vertices $(W_1, \ldots, W_\ell)$, where each vertex from U_i joins each set independently with probability $1/q$. The universe for the search is the set $X \triangleq \{G[W_i]\}_{i \in [\ell]}$, and the boolean function g is defined as $g(G[W_i]) = 1$ if $G[W_i]$ contains a triangle, and 0 otherwise.

The correctness of the algorithm follows because if the graph G has t triangles, then there exists an index $i \in [s]$ such that $G[U_i]$ contains a triangle with probability at least 1/10. The i-th team finds this triangle using $\sqrt{1/q^3}$ evaluations of g w.h.p. The final detail of the algorithm is to set q such that each evaluation of g takes $\mathcal{O}(1)$ rounds, which optimizes the round complexity of this approach.

Our above fast triangle detection algorithm for the Quantum Congested Clique model is given in the full version of the paper, as well as a discussion on why extending our approach to h-cycle detection is not straightforward.

Additional Related Work. In this Congested Clique model, matrix multiplication was first studied by [20]. After the aforementioned works of [12, 27], the work of [13] showed an algorithm whose running time improves with the sparsity of the input matrices, and [8] showed algorithms for sparse matrix multiplication which also enjoy the sparsity of the output or when only a sparse piece of the output is needed.

The task of listing subgraphs has also received great attention in the Congested Clique model. Here, each vertex needs to output a list of copies of the subgraph H, such that the union of the lists is exactly the set of all copies of H in the graph. As mentioned, [19] give an algorithm for listing all h-vertex graphs within $O(n^{1-2/h})$ rounds. For triangles, this is known to be tight by [32, 38]. This optimality extends to larger cliques due to [25]. Listing triangles in sparse graphs can be done faster, as first shown by [38] with a randomized algorithm, and then followed up by [13] with a deterministic algorithm. Afterward, [11] showed faster sparse listing for larger cliques, which also has a deterministic algorithm due to [10].

We mention that in the closely-related Congest model, in which the communication graph is the input graph itself, rather than being a complete network, the state of affairs is in stark contrast to the Congested Clique model. For listing cliques, optimal algorithms are known due to [16, 7], with deterministic solutions in [17, 14, 15]. The underlying approach, initiated by [16], is to construct an expander decomposition, which partitions the vertices into components with good expansion (low mixing time). At a very high level, the vertices of each component list cliques for which some edges are inside the component, and then recurse over the remaining edges. However, for an algorithmic approach of the Congested Clique model to have a fast implementation in the Congest model, also when using the known routing procedures [28, 29], the algorithm has to adhere to certain conditions. In other words, it is not the case that any algorithm in the Congested Clique model can be executed efficiently by the components of the expander decomposition in the Congest model.

Specifically, we stress that for the detection variant in Congest, the state of the art even just for triangles is the same as for listing. That is, even the $O(n^\rho)$ algorithm of [12] does not have an implementation in the Congest model, and it is unknown how to detect triangles in less than the time it takes for listing them (interestingly, the only lower bounds that are known are that a single round does not suffice [1, 25]). For larger h-cycles, detection for odd values of h is known to have a complexity of $\tilde{\Theta}(n)$ [21]. Much work is invest in studying the complexity of detecting even cycles [21, 34, 10, 22, 9, 39, 26], with the state of the art being a recent result showing that h-cycles can be detected in $\tilde{O}(n^{1-2/h})$ rounds for even values of h [26].

Investigations into the subgraph detection problem have also been conducted in additional models such as the quantum Congest and Quantum Congested Clique models, where vertices exchange qubits instead of standard bits. In [33], a quantum Congest $\tilde{\mathcal{O}}(n^{1/4})$-round algorithm for triangle detection was presented, which outperforms the $\tilde{\mathcal{O}}(n^{1/3})$-round Congest algorithm. This approach was further improved in [9] by developing an $\tilde{\mathcal{O}}(n^{1/5})$ rounds quantum algorithm. Additional upper and lower bounds for cycle detection in the quantum Congest model were presented in [39, 26]. In [9], an Quantum Congested Clique algorithm for p-clique detection, for $p \geq 4$, was presented, which achieves an $\tilde{\mathcal{O}}(n^{1-2/(p-1)})$-round complexity, which is faster than the classical Congested Clique algorithm. Further research in the quantum distributed models includes both upper and lower bounds for various problems [23, 31, 39].

There is extensive research about subgraph finding in additional models of distributed computing. All of these important works are a bit more far from our work here, and hence we refer the reader to the survey of [6], which contains a recent overview of subgraph finding algorithms for distributed settings.

2 Preliminaries

We use $[n]$ to denote the set $\{1, \ldots, n\}$. We denote the base graph by G, its vertices by $V(G)$, where unless stated otherwise we assume $V(G) = [n]$. Fix some constant integer $h \geq 0$. Let t denote the number of h-cycles in G, and let $V_{\mathcal{C}_h}(G)$ denote the set of vertices that participate in an h-cycle in G, where we denote by x the size of the set $V_{\mathcal{C}_h}(G)$. This parameter plays a crucial role in the analysis in Appendix A. We establish a connection between the two parameters x and t as follows.

▶ **Definition 5** (δ)**.** *For undirected graph, G with t copies of h-cycle, and $V_{\mathcal{C}_h}(G) = x$, we define δ as such that $x^{h-\delta} = 2h \cdot t$. If G is directed, we define δ as such that $x^{h-\delta} = h \cdot t$.*

We present two claims on δ. We show that $\delta \in [0, h-1]$, and that the term $x^{-\delta}$ is equal to is the probability for h vertices sampled uniformly at random with replacement from $V_{\mathcal{C}_h}(G)$ to form an h-cycle in G.

▷ Claim 6. Let G be a graph with t copies of an h-cycle and x vertices that participate in at least one h-cycle. Sample h vertices $(v_1, \ldots, v_h)$ from $V_{\mathcal{C}_h}(G)$ uniformly at random with replacement from $V_{\mathcal{C}_h}(G)$. Then the probability that they form an h-cycle is exactly $x^{-\delta}$.

▷ Claim 7. It holds that $\delta \in [0, h-1]$.

The proofs of Claims 6 and 7 are deferred to the full version of the paper.

2.1 Additional Tools

▶ **Lemma 8** (Lenzen's Routing Lemma [36])**.** *The following is equivalent to the Congested Clique model: In every round, each vertex can send (receive) $\{b_i\}_{i \in [n]}$ bits to (from) the i-th vertex, for any sequence $\{b_i\}_{i \in [n]}$ satisfying $\sum_{i=1}^{n} b_i = \mathcal{O}(n \log n)$. In other words, any routing scheme in which no vertex sends or receives more than $\mathcal{O}(n)$ messages can be preformed in $\mathcal{O}(1)$ rounds.*

▶ **Theorem 9** (Chernoff Bound [18, Corollary 1.10.6.]). *Let $X_1, \ldots, X_n$ be independent random variables taking values in $[0,1]$ and $X = \sum_i X_i$. Let $\delta \in [0,1]$. Then,* $\Pr[|X - \mathbb{E}[X] \geq \delta\mathbb{E}[X]|] \leq 2\exp\left(-\delta^2 \cdot \mathbb{E}[X]/3\right)$

▶ **Theorem 10** (Reverse Markov's inequality [18, (1.6.4)]). *Let X be a random variable with support contained in $[0,M]$. Then, for $R \in \mathbb{R}$, we have $\Pr[X > R] \geq \frac{\mathbb{E}[X]-R}{M-R}$.*

▶ **Theorem 11** (FMM-based triangle detection [12]). *There is a deterministic algorithm for triangle detection, which takes $\mathcal{O}(n^\rho)$ rounds.*

3 Fast Matrix Multiplication in Congested Clique

3.1 Preliminaries and Balanced Products

In this section, we define the problem of computing s pairs of square matrices of size k each, as well as defining what is a balance input. We assume throughout the paper that the matrices are over a field $\mathbb{F}$, where each element can be represented using $\mathcal{O}(\log n)$ bits. Due to space constraints, we the proofs of this section are omitted, and can be found in the full version of the paper. We introduce the following definitions to specify the required input.

▶ **Definition 12** (The notations $A[i,*]$ and $A[*x*,*y*]$). *Let A be some matrix. We denote the i-th row of A by $A[i,*]$. For a matrix A of dimension $n \times n$ and two indices $x, y \in [\sqrt{k}]$ for $k \leq n$, we also use $A[*x*,*y*]$ to denote a matrix of dimension $n/\sqrt{k} \times n/\sqrt{k}$, which is the following submatrix of A. For every index $v \in [n]$, we split it into three indices $v = v_1v_2v_3$ where $v_1, v_3 \in [n^{1/2} \cdot k^{-1/4}]$, $v_2 \in [\sqrt{k}]$. The expression $*x*$ then refers to all v for which $v_2 = x$.*

The following definition formally defines the problem of multiple matrix multiplications.

▶ **Definition 13** (Product $(\mathcal{Q})$). *Given set of s pairs of square matrices $\mathcal{Q} = \{(S_i, T_i)\}_{i\in[s]}$ of size $k \times k$. In the* Product $(\mathcal{Q})$ *problem, n nodes need to compute the products of those pairs of matrices. The input is distributed as follows. Each vertex $v \in V$ is assigned a label $\ell(v) = (i, x, y)$, where $i \in [s]$, and $x, y \in [\sqrt{n/s}]$. The vertex v gets as input the submatrix $S_i[*x*,*y*], T_i[*x*,*y*]$, and has to learn the entries of the submatrix $P_i[*x*,*y*]$, where $P_i = S_i \cdot T_i$ for $i \in [s]$. We denote the round complexity of this problem by* $\mathsf{MM}(k,k,k;s)$.

Note that every vertex can learn the label of each other vertex in $\mathcal{O}(1)$ rounds. The following theorem is the main theorem for this section, in which we provide an upper bound for the round complexity of the Product $(\mathcal{Q})$ problem.

▶ **Theorem 14.** *For any two integers k, s where $k \in [\sqrt{n}, n]$ and $s \leq (n/k)^2$, we have that*

$$\mathsf{MM}(k,k,k;s) = \mathcal{O}(n^{\rho-2} \cdot k^2 \cdot s^{1-\rho}).$$

To prove the theorem, we first partition the n nodes into s sets of size n/s each. For every $i \in [s]$ we call the i-th set in the partition the i-th *team*. The i-th team is responsible for computing the i-th product in $\mathcal{Q}$, i.e., the product (S_i, T_i). After partitioning into teams, the problem boils down to computing one product of square matrices of size k using n/s node with bandwidth of size $s \log n$. This extends [12], in which only the product of square matrices of size equal to the number of vertices is considered, and uses [27], in which multiple products are divided into teams. A crucial step in the algorithm for Theorem 14 is to compute the product of a single matrix of size R using n' nodes, which we define next.

▶ **Definition 15** (Single-Product (n', R)). *Let H be a team with n' vertices. Let S, T be two square matrices of dimension R for some $R \in [1, (n')^2]$, and define $P = ST$. Each vertex $v \in H$ is assigned a label $\ell'(v) = xy$, where $x, y \in [\sqrt{n'}]$. The input of each vertex $v \in H$ with label $\ell'(v) = xy$ is $S[*x*, *y*]$ and $T[*x*, *y*]$, and its output should be $P[*x*, *y*]$. We denote this problem by* Product (n', R).

▶ **Proposition 16.** *The* Single-Product (n', R) *problem can be solved in the* Congested Clique *model with n' vertices in the base graph and bandwidth B, in $\mathcal{O}((n')^\rho \cdot (R/n')^2 \cdot \frac{F}{B})$ rounds, where each entry in R can be represented using $\mathcal{O}(F)$ bits. Using bandwidth B means that in each round, each vertex in the base graph can send B bits to every other vertex.*

3.2 Multiple Products of Random Submatrices

In this subsection, we explain how to use the tools we developed in the previous subsection, to detect an h-cycle in s induced subgraphs sampled uniformly at random. Specifically, we explain how to compute the h-th power of the adjacency matrices of those subgraphs.

Let $\mathcal{U} = (U_1, \ldots, U_s)$ be a set of subsets of vertices, where each subset is a uniformly random set. That is, each vertex joins to the set U_i independently and uniformly at random, with probability p. For each $i \in [s]$, we denote the the adjacency matrix of $G[U_i]$ by A_i, and define $\mathcal{Q} = \{(A_i, A_i)\}_{i \in [s]}$.

We explain how to compute the h-th power of $\{A_i\}_{i \in [s]}$ in parallel by reducing this problem into the Product$(\mathcal{Q})$ problem. In other words, we explain how to redistribute the initial input, into an input for the Product$(\mathcal{Q})$ problem. We show that the reduction takes $\mathcal{O}(\log n)$ rounds (Proposition 20) w.h.p., and provide an algorithm that tests whether the reduction algorithm can be executed in $\mathcal{O}(\log n)$ or not (Claim 23). The testing algorithm takes $\mathcal{O}(1)$ rounds. In case the testing algorithm indicates that we sampled a set $\mathcal{U}$ for which the reduction takes more than $\mathcal{O}(\log n)$ rounds, we discard the current sample set $\mathcal{U}$, and sample a new set. We also prove that w.h.p. we will not have to discard the sampled set (Claim 22).

In Section 3.1, we described an algorithm to compute the product of s pairs of matrices of size k each. Here, we describe an algorithm to compute the product of $1/p^a$ pairs of matrices of size *at most* $4np$ each. The connection between the parameters s, k, p and a is as follows. We set $k = 4np$, and $s = 1/p^a$ where $a \in [0, 2]$. We get that $sk^2 \le n^2$ as desired. We provide a definition for a set $\mathcal{U}$ for which we can redistribute the input in $\mathcal{O}(\log n)$ rounds. We call such a set a *p-balanced* set.

▶ **Definition 17** (p-Balanced Set). *Given is a parameter p. Let $\mathcal{U}$ be a set of subsets of vertices from $V(G)$. Let a be a constant for which $|\mathcal{U}| = p^{-a}$. We say that $\mathcal{U}$ is a p-*balanced set *if all the following conditions hold:*

1. *$a \in [0, 2]$ (so $|\mathcal{U}| \le (1/p)^2$).*
2. *$n^{-1/2} \le p \le 1$.*
3. *Every vertex $v \in V(G)$ belongs to at most $\lceil |\mathcal{U}| \cdot p \rceil 4 \log n = \lceil p^{1-a} \rceil 4 \log n$ sets in $\mathcal{U}$.*
4. *Every set $U \in \mathcal{U}$ is of size at most $\lceil 4np \rceil$.*

Note that (1) and (2) imply that $|\mathcal{U}| \le n$.

The next claim proves that if $\mathcal{U} = (U_1, \ldots, U_{p^{-a}})$ is a p-balanced set, then every vertex v can learn the IDs of all vertices in U_j for each set U_j to which v belongs.

▷ Claim 18. Let $\mathcal{U} = (U_1, \ldots, U_{p^{-a}})$ be a p-balanced set, where every vertex knows to which U_j it belongs. There is an $\mathcal{O}(\log n)$-round Congested Clique algorithm that allows each vertex to learn the IDs of all vertices in U_j for each set U_j to which v belongs.

In what follows, we explain how to route the input of a p-balanced set $\mathcal{U}$, after each vertex learns the IDs of all vertices in U_j for each set U_j to which v belongs, to match the input of the $\mathsf{Product}(\mathcal{Q})$ problem. This routing takes $\mathcal{O}(\log n)$ rounds. Before providing a routing algorithm, we introduce new notation that we need in order to explain how the input is routed.

▶ **Definition 19** (The notation $A[i, U_j]$). *Recall that $V = [n]$, and let $U_j \subset V$, and let i be some vertex in U_j. Let A_j be the corresponding adjacency matrix of U_j. For vertex i in the set U_j we define $A[i, U_j]$ as the submatrix of A, which contains only the i-th row, and all k columns, for $k \in U_j$.*

▶ **Proposition 20** (Redistributing the Input). *Given a parameter p, let $\mathcal{U} \triangleq (U_1, \dots, U_{p^{-a}})$ be a set of subsets of vertices from $V(G)$ which is a p-balanced set. For each $i \in [p^{-a}]$, let A_i be the adjacency matrix of the induced graph $G[U_i]$. Label each vertex $v \in [n]$ as $\ell(v) \triangleq (x, y, i) \in [\sqrt{np^a}] \times [\sqrt{np^a}] \times [p^{-a}]$. Partition the vertices into p^{-a} teams, each of size $n \cdot p^a$, where the j-th team contains all vertices v with label $\ell(v) = (x, y, i)$ such that $i = j$. Then, in parallel, each vertex v with label $\ell(v) = (x, y, i)$ can learn $A_i[*x*, *y*]$ in $\mathcal{O}(\log n)$ rounds.*

The next corollary address the detection of an h-cycle in one of the sampled graphs. It follows from Proposition 20 and Theorem 14. The algorithmic aspects of this corollary are presented in Appendix A.

▶ **Corollary 21.** *Given r and p, let $\mathcal{U} = (U_1, \dots, U_r)$ be a set of subsets of vertices from $V(G)$, where $\mathcal{U}$ is a p-balanced set, and every vertex in G knows whether it belongs to U_j for every $j \in [s]$. For each $i \in [p^{-a}]$, let A_i be the adjacency matrix of the induced graph $G[U_i]$. Label each vertex $v \in [n]$ as $\ell(v) \triangleq (x, y, i) \in [\sqrt{np^a}] \times [\sqrt{np^a}] \times [p^{-a}]$. Then, for every integer h, in parallel, each vertex v with label $\ell(v) = (x, y, i)$ can learn $(A_i)^h[*x*, *y*]$ in $\mathcal{O}(\log(h) \cdot n^\rho \cdot p^{2+a(\rho-1)} + \log n)$ rounds.*

The remainder of this subsection shows that a set $\mathcal{U}$ of uniformly random subsets of vertices is a p-balanced set w.h.p. We also provide an algorithm to test whether a set of subsets of vertices is a p-balanced set in a constant number of rounds. We create $s \triangleq p^{-a}$ subsets of vertices, by letting each vertex join each set independently with probability p (each vertex knows p and a). We denote the sets by $\mathcal{U} = (U_1, \dots, U_s)$, and show that $\mathcal{U}$ is p-balanced w.h.p., and that the vertices can determine whether this is the case in $\mathcal{O}(1)$ rounds. If $\mathcal{U}$ is indeed p-balanced then in $\mathcal{O}(\log n)$ rounds each vertex can learn the IDs of all vertices in each set U_j to which it belongs. This is

▷ Claim 22. $\mathcal{U}$ is balanced with probability at least $1 - 2/n^3$.

▷ Claim 23. There is a $\mathsf{Congested\ Clique}$ algorithm that decides if $\mathcal{U}$ is p-balanced in $\mathcal{O}(1)$ rounds.

3.3 Rectangular Matrices

Here we set the ground for computing the product of two rectangular matrices in Congested Clique. We build on the work of [27], which shows that computing the product of two rectangular matrices S, T of size $n \times n^{\beta_0}$ and $n^{\beta_0} \times n$ respectively takes $\mathcal{O}(n^{o(1)})$ rounds.

▶ **Definition 24** (Rectangular matrix multiplication $\mathsf{RM}\,(S,T)$)**.** *Given two matrices S,T of dimension $n \times n^z$ and $n^z \times n$ where $z \in [0,1]$, the* $\mathsf{RM}\,(S,T)$ *problem is to compute $P = S \cdot T$ in the Congested Clique model with n nodes. The input of each vertex $i \in [n]$ is $S[i,*]$ and $T[*,i]$, and its output should be $P[i,*]$. We abuse the notation and use it also to denote the complexity of the problem by* $\mathsf{MM}\,(n,n,n^z)$ *or* $\mathsf{RM}\,(n^z)$.

▶ Remark 25. For two matrices S,T of size $n^z \times n$ and $n \times n$ (or $n \times n$ and $n \times n^z$), we get the same round complexity [24, Theorem 6]. In this case, we assume the input is of each vertex $i \in [n]$ is $S[*,i]$ and $T[*,i]$, and its output should be $P[i,*]$.

▶ **Definition 26** (The exponent of matrix multiplication)**.** *The exponent of the* sequential *complexity of computing the product of two matrices of dimensions $n \times n^z$ and $n^z \times n$ respectively is denoted by $\omega(z)$. We denote by $\mathcal{O}(n^{\rho(z)})$ the round complexity of computing this product in the Congested Clique model. Let $\alpha_0 = \lim_{\varepsilon\to 0}\sup\{z \mid \omega(z) \le 2+\varepsilon\}$, and $\beta_0 = \lim_{\varepsilon\to 0}\sup\{z \mid \rho(z) = \varepsilon\}$. Then $\alpha_0 \geq 0.321334$ [40] and $\beta_0 \geq (1+\alpha_0)/2 \geq 0.660667$ [27, 40].*

We would like to upper bound the function $\rho(z)$ by some analytic function, which is easy to work with. To do so, we use the following notation.

▶ **Definition 27** (The notation $\mathtt{B}$, $\mathtt{A}$)**.** *We will use $\mathtt{B}, \mathtt{A}$ for two real non-negative constants such that, for every $y \in [0, 1-\beta_0]$ we have $\rho(1-y) \leq \mathtt{B} - \mathtt{A}y$.*

We give two explicit linear functions which bound the function $\rho(1-y)$. First, if the function $\rho(z)$ is convex, then we can set $\mathtt{A} = \rho(1)/(1-\beta_0)$ and $\mathtt{B} = \rho(1)$, by taking the line passing through the points $(\beta_0, \rho(\beta_0))$ and $(1, \rho(1))$. This is of course the "best" (minimizing l_∞ norm) linear function that upper bounds the function $\rho(1-y)$. Yet, proving that $\rho(z)$ is convex is beyond the scope of this paper. Instead, the following claim is an additional explicit linear function we provide, which does not assume that $\rho(z)$ is convex.

▷ Claim 28. We can set $\mathtt{A} = 0.4617$ and $\mathtt{B} = 0.1567$.

The proof of Claim 28 (appears in the full version) is numeric: We build a step function which is always above $\rho(z)$, and then find a line which is above the step function in the desired range. For any choice of $\mathtt{B}, \mathtt{A}$ that fits Definition 27 and any $p \in (0,1)$ we have $\mathsf{RM}\,(np) = \mathcal{O}(n^{\rho(1-\log_n(1/p))}) \leq \mathcal{O}(n^{\mathtt{B}-\mathtt{A}\cdot\log_n(1/p)}) = \mathcal{O}(n^{\mathtt{B}} \cdot p^{\mathtt{A}})$. Thus, by the above discussion and by Claim 28 we get the following.

▶ **Conclusion 1.** *For $p \in (0,1)$ we have* $\mathsf{RM}\,(np) \leq \mathcal{O}(n^{0.1567} \cdot p^{0.4617})$ *. If $\rho(z)$ is convex, we have* $\mathsf{RM}\,(np) \leq \mathcal{O}(n^{\rho} \cdot p^{\rho/(1-\beta_0)})$.

References

1 Amir Abboud, Keren Censor-Hillel, Seri Khoury, and Christoph Lenzen. Fooling views: a new lower bound technique for distributed computations under congestion. *Distributed Comput.*, 33(6):545–559, 2020. doi:10.1007/S00446-020-00373-4.

2 Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. More asymmetry yields faster matrix multiplication, 2024. arXiv:2404.16349, doi:10.48550/arXiv.2404.16349.

3 Noga Alon and Joel H Spencer. *The probabilistic method.* John Wiley & Sons, 2016.

4 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):844–856, 1995. doi:10.1145/210332.210337.

5 Jan van den Brand. Complexity term balancer. `www.ocf.berkeley.edu/~vdbrand/complexity/`. Tool to balance complexity terms depending on fast matrix multiplication.

6 Keren Censor-Hillel. Distributed subgraph finding: Progress and challenges (invited talk). In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 3:1–3:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. Up-to-date version on Arxiv, https://doi.org/10.48550/arXiv.2203.06597, 2021. `doi:10.4230/LIPIcs.ICALP.2021.3`.

7 Keren Censor-Hillel, Yi-Jun Chang, François Le Gall, and Dean Leitersdorf. Tight distributed listing of cliques. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2878–2891. SIAM, 2021. `doi:10.1137/1.9781611976465.171`.

8 Keren Censor-Hillel, Michal Dory, Janne H. Korhonen, and Dean Leitersdorf. Fast approximate shortest paths in the congested clique. *Distributed Comput.*, 34(6):463–487, 2021. `doi:10.1007/s00446-020-00380-5`.

9 Keren Censor-Hillel, Orr Fischer, François Le Gall, Dean Leitersdorf, and Rotem Oshman. Quantum distributed algorithms for detection of cliques. In Mark Braverman, editor, *13th Innovations in Theoretical Computer Science Conference, ITCS 2022, January 31 - February 3, 2022, Berkeley, CA, USA*, volume 215 of *LIPIcs*, pages 35:1–35:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPICS.ITCS.2022.35`.

10 Keren Censor-Hillel, Orr Fischer, Tzlil Gonen, François Le Gall, Dean Leitersdorf, and Rotem Oshman. Fast distributed algorithms for girth, cycles and small subgraphs. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 33:1–33:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPICS.DISC.2020.33`.

11 Keren Censor-Hillel, François Le Gall, and Dean Leitersdorf. On distributed listing of cliques. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 474–482, 2020. `doi:10.1145/3382734.3405742`.

12 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. *Distributed Comput.*, 32(6):461–478, 2019. `doi:10.1007/s00446-016-0270-2`.

13 Keren Censor-Hillel, Dean Leitersdorf, and Elia Turner. Sparse matrix multiplication and triangle listing in the congested clique model. *Theoretical Computer Science*, 809:45–60, 2020. `doi:10.1016/J.TCS.2019.11.006`.

14 Keren Censor-Hillel, Dean Leitersdorf, and David Vulakh. Deterministic near-optimal distributed listing of cliques. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 271–280, 2022. `doi:10.1145/3519270.3538434`.

15 Yi-Jun Chang, Shang-En Huang, and Hsin-Hao Su. Deterministic expander routing: Faster and more versatile. *arXiv preprint arXiv:2405.03908*, 2024. `doi:10.48550/arXiv.2405.03908`.

16 Yi-Jun Chang, Seth Pettie, Thatchaphol Saranurak, and Hengjie Zhang. Near-optimal distributed triangle enumeration via expander decompositions. *Journal of the ACM (JACM)*, 68(3):1–36, 2021. `doi:10.1145/3446330`.

17 Yi-Jun Chang and Thatchaphol Saranurak. Deterministic distributed expander decomposition and routing with applications in distributed derandomization. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 377–388. IEEE, 2020. `doi:10.1109/FOCS46700.2020.00043`.

18 Benjamin Doerr and Frank Neumann, editors. *Theory of Evolutionary Computation - Recent Developments in Discrete Optimization*. Natural Computing Series. Springer, 2020. `doi:10.1007/978-3-030-29414-4`.

19 Danny Dolev, Christoph Lenzen, and Shir Peled. “tri, tri again”: finding triangles and small subgraphs in a distributed setting. In *Distributed Computing: 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings 26*, pages 195–209. Springer, 2012.

20 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 367–376. ACM, 2014. doi:10.1145/2611462.2611493.

21 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 367–376, 2014. doi:10.1145/2611462.2611493.

22 Talya Eden, Nimrod Fiat, Orr Fischer, Fabian Kuhn, and Rotem Oshman. Sublinear-time distributed algorithms for detecting small cliques and even cycles. *Distributed Computing*, pages 1–28, 2022.

23 Michael Elkin, Hartmut Klauck, Danupon Nanongkai, and Gopal Pandurangan. Can quantum communication speed up distributed computation? In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 166–175. ACM, 2014. doi:10.1145/2611462.2611488.

24 Michael Elkin and Ofer Neiman. Centralized, parallel, and distributed multi-source shortest paths via hopsets and rectangular matrix multiplication. In *39th International Symposium on Theoretical Aspects of Computer Science (STACS 2022)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2022.

25 Orr Fischer, Tzlil Gonen, Fabian Kuhn, and Rotem Oshman. Possibilities and impossibilities for distributed subgraph detection. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 153–162, 2018. doi:10.1145/3210377.3210401.

26 Pierre Fraigniaud, Mael Luce, Frederic Magniez, and Ioan Todinca. Even-cycle detection in the randomized and quantum congest model. *arXiv preprint arXiv:2402.12018*, 2024.

27 François Le Gall. Further algebraic algorithms in the congested clique model and applications to graph-theoretic problems. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, volume 9888 of *Lecture Notes in Computer Science*, pages 57–70. Springer, 2016. doi:10.1007/978-3-662-53426-7_5.

28 Mohsen Ghaffari, Fabian Kuhn, and Hsin-Hao Su. Distributed mst and routing in almost mixing time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2017. doi:10.1145/3087801.3087827.

29 Mohsen Ghaffari and Jason Li. New distributed algorithms in almost mixing time via transformations from parallel algorithms. In *32nd International Symposium on Distributed Computing*, 2018.

30 Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996. doi:10.1145/237814.237866.

31 Taisuke Izumi and François Le Gall. Quantum distributed algorithm for the all-pairs shortest path problem in the CONGEST-CLIQUE model. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 84–93. ACM, 2019. doi:10.1145/3293611.3331628.

32 Taisuke Izumi and François Le Gall. Triangle finding and listing in congest networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 381–389, 2017. doi:10.1145/3087801.3087811.

33 Taisuke Izumi, François Le Gall, and Frédéric Magniez. Quantum distributed algorithm for triangle finding in the congest model. In *37th International Symposium on Theoretical Aspects of Computer Science (STACS 2020)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

34 Janne H. Korhonen and Joel Rybicki. Deterministic subgraph detection in broadcast CONGEST. In James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão, editors, *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal,*

December 18-20, 2017, volume 95 of *LIPIcs*, pages 4:1–4:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPICS.OPODIS.2017.4`.

35 François Le Gall and Frédéric Magniez. Sublinear-time quantum computation of the diameter in congest networks. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2018. URL: `https://dl.acm.org/citation.cfm?id=3212744`.

36 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 42–50, 2013. `doi:10.1145/2484239.2501983`.

37 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in o (log log n) communication rounds. *SIAM journal on computing*, 35(1):120–131, 2006. `doi:10.1137/S0097539704441848`.

38 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the distributed complexity of large-scale graph computations. *ACM Transactions on Parallel Computing (TOPC)*, 8(2):1–28, 2021. `doi:10.1145/3460900`.

39 Joran van Apeldoorn and Tijn de Vos. A framework for distributed quantum queries in the CONGEST model. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 109–119. ACM, 2022. `doi:10.1145/3519270.3538413`.

40 Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proc. SODA*, page to appear, 2024.

A h-Cycle Detection

In this section, we prove the following theorem.

▶ **Theorem 1** (h-Cycle Detection). *Let G be a (directed) graph with t copies of h-cycles. There is a randomized Congested Clique algorithm for h-cycle detection, which takes $\tilde{O}(h^{O(h)} \cdot n^{0.1567}/(t^{\frac{0.4617}{h-1.82408}}+1))$ rounds w.h.p.*

We do so by presenting two algorithms and analyzing their running time and success probability as a function of the parameters n, t, and x. Figure 2 depicts their running times. All the proofs, and implementation details are omitted due to space constraints and can be found in the full version of the paper.

Before presenting the algorithms, we overview the color-coding technique [4], which is a common method used to find paths or cycles of constant length h. To detect an h-cycle, first color the vertices of the graph using h colors, where each vertex is colored uniformly at random and independently of all other vertices. Then look for a *colorful* h-cycle, which is an h-cycle with exactly one vertex of each color. This provides additional structure, which a detection algorithm can benefit from. However, not every coloring induces a colorful h-cycle, which means that to detect an h-cycle, we might have to repeat this experiment multiple times, until we sample a coloring that induces a colorful h-cycle.

In more detail, given a graph G with n vertices, we sample a uniformly random coloring $\varphi : V \to [h]$, which means that φ colors each vertex uniformly independently at random. We then build the auxiliary graph G_φ which is a directed graph, as follows.

▶ **Definition 29** (G_φ). *Given a graph G, and a coloring $\varphi : V \to [h]$, we define a new directed graph G_φ on the same vertex set, with a set of directed edges $E(G_\varphi) \triangleq \{(u,v) \in E(G) \mid \varphi(v) = (\varphi(u)+1) \mod h\}$. That is, only a subset of the edges is kept, and it consists of the edges from the vertices of color i to the vertices of color $(i+1) \mod h$, for $i \in [h]$.*

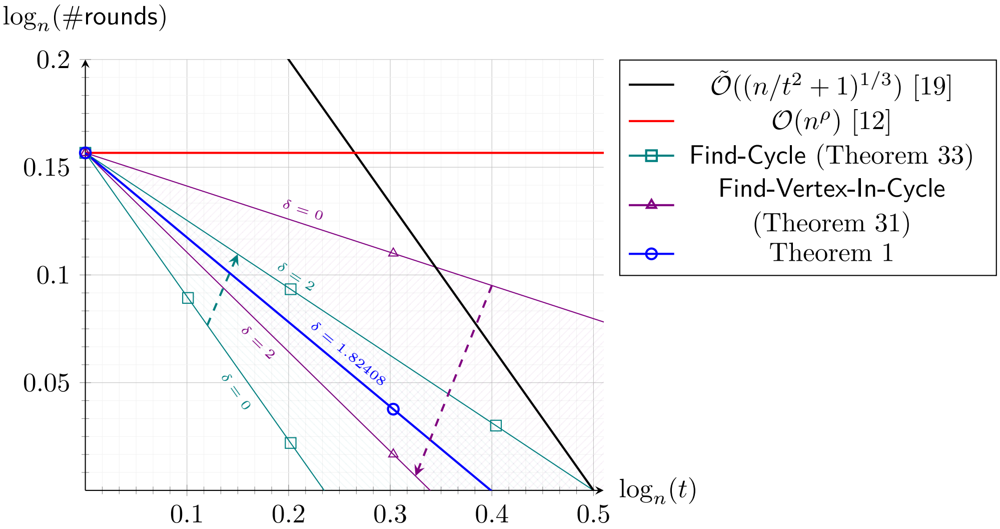


Figure 2 An illustrative comparison between our results and prior work, for the case of triangles. For each algorithm, we plot the base-n logarithm of the number of rounds as a function of the base-n logarithm of the number of triangles. An additional axis represents the value of δ ranging from 0 to 2. For a fixed t, Find-Cycle performs faster as δ decreases, with its round complexity depicted by the area shaded in teal. Conversely, Find-Vertex-In-Cycle performs better as δ increases, and its round complexity is shown by the area shaded in violet.

The graph G_φ has the property that every walk of length smaller than h is a simple path, and every closed walk of length h is a cycle. Here, a *walk* of length h on a (directed) graph is a sequence of vertices $(v_1, v_2, \dots, v_{h+1})$ not necessarily distinct, such that for $i \in [h]$ we have that (v_i, v_{i+1}) is an edge in G. We say that a walk is a simple path if all the vertices in the walk are distinct. A walk $(v_1, v_2, \dots, v_h, v_{h+1})$ is *closed* if $v_1 = v_{h+1}$.

We explain how we benefit from the property that every closed walk of length h in G_φ is a cycle. Let A_φ denote the adjacency matrix of G_φ. We can compute the h-th power of the matrix A_φ, and look at the diagonal of the obtained matrix. Then, G_φ is h-cycle free if and only if all the entries on the diagonal are equal to 0. Clearly, if G does not contain an h-cycle, then for any coloring φ, we have that G_φ does not contain an h-cycle. The more interesting property of this random coloring is that if G contains an h-cycle, then the probability that G_φ contains one is at least $1/h^h$, as we prove next.

▷ Claim 30. Let G be a graph with at least one h-cycle. Let $\varphi : V \to [h]$ be some uniformly random coloring. Then G_φ contains an h-cycle with probability at least $\frac{1}{h^h}$.

A.1 The Algorithm Find-Vertex-In-Cycle

We explain how to detect an h-cycle in time $\mathcal{O}(\mathsf{MM}\,(n, n, n/x) \cdot \log^2 n)$ w.h.p., with a one-sided error, as stated in the next theorem.

▶ **Theorem 31.** *There exists a randomized Congested Clique algorithm to detect an h-cycle in time $\tilde{\mathcal{O}}(\mathsf{MM}\left(n, n, \frac{n}{x}\right))$ w.h.p., with a one-sided error.*

Let G be a graph with n vertices and t copies of an h-cycle, for a fixed constant h. For a graph H, we denote by $V_{\mathcal{C}_h}(H)$ the set of vertices that participate in an h-cycle in H. Let $x = |V_{\mathcal{C}_h}(G)|$. We prove Theorem 31 by analyzing the following random process.

Find-Vertex-In-Cycle. The input of the algorithm is a graph G and some value $p \in [0,1]$. The output is "True" if at least one h-cycle is detected, and "False" otherwise. The algorithm works as follows. The algorithm samples a coloring $\varphi : V(G) \to [h]$, uniformly at random, and uses it to define a new auxiliary graph G_φ, as explained in Definition 29. Let V_i denote the set of vertices in G_φ that are assigned the color i, for $i \in [h]$. The algorithm then samples a subset of vertices from V_1, by sampling each vertex independently with probability p. Let U_1 denote the set obtained. Define F_φ as the induced subgraph of G_φ with the vertex set $U_1 \cup \bigcup_{i=2}^{h} V_i$. Let A_{F_φ} denote the adjacency matrix of the graph F_φ. Next, the algorithm exactly counts the number of h-cycles in F_φ using rectangular matrix multiplication. That is, it computes the trace of the h-th power of A_{F_φ}, and outputs "True" if it is not zero, and "False" otherwise. Clearly, this can be computed by first computing the h-th power of A_{F_φ}, and then computing its trace, which takes $\mathcal{O}(\mathsf{MM}(n,n,n))$ rounds. However, a faster well-known way to compute this trace, without computing the h-th power of A_{F_φ}, is as follows. Compute the following product:

$$A_{F_\varphi}[U_1, V_2] \cdot A_{F_\varphi}[V_2, V_3] \cdots A_{F_\varphi}[V_{h-1}, V_h] \cdot A_{F_\varphi}[V_h, U_1] \ ,$$

where for $S, T \subseteq V$ the matrix $A_{F_\varphi}[S,T]$ denotes the rectangular matrix with $|S|$ rows and $|T|$ columns, every for every $s \in S$ and $T \in t$ we have that $(A_{F_\varphi}[S,T])_{s,t} = 1$ if $(s,t) \in E(F_\varphi)$ and 0 otherwise. This matrix is also called the biadjacency matrix. The order in which the multiplications are computed affects the round complexity. The algorithm computes this product by sequentially multiplying a rectangular matrix of size at most $4np \times n$ and a matrix of size at most $n \times n$, to get a new matrix of size $4np \times n$. In other words, the algorithm first computes the product $A_{F_\varphi}[U_1, V_2] \cdot A_{F_\varphi}[V_2, V_3]$, to obtain some matrix B_2, and then computes the product $B_2 \cdot A_{F_\varphi}[V_3, V_4]$. In this way, the algorithm does not multiply two square matrices of size n, and can benefit from the fact that it only computes the product of one smaller rectangular matrix with a square one. This completes the description of the algorithm.

Clearly, the algorithm never outputs "True" if the graph G is h-cycle free. In what follows, we give a lower bound on the probability that it outputs "True" when the graph has h-cycles.

▷ Claim 32. If the sampling probability of vertices from V_1 into U_1 satisfies $p \geq \frac{4h^h}{x}$, then the algorithm outputs "True" with probability at least $\frac{1}{4h^h}$.

The implementation of the algorithm in the Congested Clique model appears in the full version of the paper.

A.2 The Algorithm Find-Cycle

In the next two subsections, we explain how to prove the following theorem.

▶ **Theorem 33.** *There exists a randomized Congested Clique algorithm to detect an h-cycle in time $\tilde{\mathcal{O}}(\mathsf{MM}\left(\frac{n}{x}, \frac{n}{x}, \frac{n}{x}; x^\delta\right))$ w.h.p., with one-sided error.*

Recall that G is a graph with n vertices and t copies of an h-cycle for $h = \mathcal{O}(1)$. For a graph H, we denote by $V_{\mathcal{C}_h}(H)$ the set of vertices that participates in an h-cycle in H. Let $x = |V_{\mathcal{C}_h}(G)|$. We also use δ for the solution for $x^{h-\delta} = 2ht$ satisfies that $\delta \in [0, h-1]$.

▶ Remark 34. Recall that $\mathsf{MM}\left(\frac{n}{x}, \frac{n}{x}, \frac{n}{x}; x^\delta\right) = \mathcal{O}(n^\rho \cdot x^{-(2+\delta(\rho-1))})$, by Corollary 21.

We prove Theorem 33 by analyzing the following random process.

Find-Cycle. The input of the algorithm is a graph G A graph G, a value $p \in [0,1]$, and a value $a \in [0,2]$. The output is "True" if at least one h-cycle is detected, and "False" otherwise. The algorithm works as follows.

1. Sample uniformly at random a coloring $\varphi : V \to [h]$.
2. Sample $r \leftarrow 8(4h)^{h+2} \cdot p^{-a}$ subsets of vertices $\mathcal{U} = (U_1, \ldots, U_r)$, where each vertex joins U_i independently with probability p for $i \in [r]$.
3. For every $U \in \mathcal{U}$, define two graphs. The first one is the induced graph $F = G[U]$, and the second one is the colored directed graph F_φ, obtained from applying φ on F. Denote the adjacency matrix of F_φ by M_U.
4. For $U \in \mathcal{U}$, compute the trace of the h-th power of the matrix M_U, and output "True" if for at least one set U, this trace is not zero. Otherwise, output "False".

Fix some set $U \in \mathcal{U}$, and a random coloring φ, and let $F = G[U]$, and $F_\varphi = (G[U])_\varphi$. We prove that for $p \geq 1/x$, the subgraph F_φ contains an h-cycle with probability $\Omega\left(x^{-\delta}\right)$. For that, it suffices to prove that if $p \geq 1/x$ then F contains an h-cycle with probability at least $\frac{1}{x^\delta \cdot (4h)^{h+1}}$: We proved in Claim 30 that if F contains an h-cycle then F_φ contains an h-cycle with probability at least $\frac{1}{h^h}$. In the next proposition, we prove that the subgraph F contains an h-cycle with probability at least $\frac{1}{x^\delta \cdot (4h)^{h+2}}$, if $p \geq \frac{1}{x}$.

▶ **Proposition 35.** *If $p \geq \frac{1}{x}$, then F contains an h-cycle with probability at least $\frac{1}{x^\delta \cdot (4h)^{h+2}}$.*

To prove the above, we use the second moment method [3, Theorem 4.3.1]. The proof of the proposition, as well as the implementation of the algorithm in the Congested Clique model, is deferred to the full version of the paper.

A.3 Wrap-Up: Fast Cycle Detection

In this subsection, we wrap up to prove our fast algorithm for h-cycle detection, when $h = \mathcal{O}(1)$, in both undirected and directed graphs. Our algorithm is the fastest for odd cycle detection when the number of cycles is super polylogarithmic, and for h-cycle detection in directed graphs, when the number of h-cycles is super polylogarithmic. For graphs with small t, our algorithm has the same running time as the fastest algorithm for multiplying two matrices of size $n \times n$, and our running time is never worse than it up to polylogarithmic factors.

▶ **Theorem 1** (h-Cycle Detection)**.** *Let G be a (directed) graph with t copies of h-cycles. There is a randomized Congested Clique algorithm for h-cycle detection, which takes $\tilde{\mathcal{O}}(h^{\mathcal{O}(h)} \cdot n^{0.1567}/(t^{\frac{0.4617}{h-1.82408}}+1))$ rounds w.h.p.*

Let $\mathcal{R}(G)$ denote the round complexity of Theorem 1. Let $\mathcal{R}_1(G), \mathcal{R}_2(G)$ denote the round complexity of the algorithms in Theorem 33 and Theorem 31 respectively. To prove Theorem 1, we show that for every graph G with n vertices and t copies of an h-cycle, we have $\min\{\mathcal{R}_1(G), \mathcal{R}_2(G)\} \leq \mathcal{R}(G)$. To show that, we use a case analysis. Recall that x denotes the number of vertices in G that participate in an h-cycle, and that $x^{h-\delta} = 2ht$. We show that if $\delta \geq 1.82408$, then $\mathcal{R}_2(G) \leq \mathcal{R}(G)$, and if $\delta \leq 1.82408$, then $\mathcal{R}_1(G) \leq \mathcal{R}(G)$.

The theorem then follows, as we can run the algorithms Find-Cycle and Find-Vertex-In-Cycle one step at a time, until one of them detects a triangle.

Proof of Theorem 1.

The Case $\delta \geq 1.82408$. The execution of the algorithm Find-Vertex-In-Cycle takes $\mathsf{MM}\left(\frac{n}{x}, n, n\right)$ rounds, where $\mathsf{MM}\left(\frac{n}{x}, n, n\right) \leq n^{\mathtt{B}} \cdot x^{-\mathtt{A}}$ by Definition 27. Since we assumed that $\delta \geq 1.82408$, we have $x = (2ht)^{1/(h-\delta)} \geq t^{1/(h-1.82408)}$. We get that $\mathcal{R}_2(G) \leq n^{\mathtt{B}} \cdot t^{-\frac{\mathtt{A}}{2-1.82408}}$. By plugging in $\mathtt{A} = 0.4617, \mathtt{B} = 0.1567$, (see Claim 28) we get that the round complexity is bounded by $n^{0.1567} \cdot t^{-\frac{0.4617}{2-1.82408}}$, which completes the proof of this case.

The Case $\delta \leq 1.82408$. The execution of the algorithm Find-Vertex-In-Cycle takes $\mathsf{MM}\left(\frac{n}{x}, \frac{n}{x}, \frac{n}{x}; x^{\delta}\right)$ rounds, where

$$\begin{aligned}
\mathsf{MM}\left(\frac{n}{x}, \frac{n}{x}, \frac{n}{x}; x^{\delta}\right) =& \mathcal{O}(n^{\rho}/x^{2+\delta(\rho-1)}) \\
=& \mathcal{O}(n^{\rho}/t^{\frac{2+\delta(\rho-1)}{h-\delta}}) \\
\leq& \mathcal{O}(n^{\rho}/t^{\frac{2+1.82408(\rho-1)}{h-1.82408}}) \\
\leq& \mathcal{O}(n^{0.1567} \cdot t^{\frac{0.4617}{h-1.82408}})
\end{aligned}$$

The first equality follows from Theorem 14. The penultimate inequality follows since the function $\delta \mapsto \frac{2+\delta(\rho-1)}{h-\delta}$ is monotonically decreasing in the range $\delta \in [0, 1.82408]$. The last inequality follows by setting $\rho \leftarrow 0.1567$, which completes the proof. ◀

Deterministic Self-Stabilising Leader Election for Programmable Matter with Constant Memory

Jérémie Chalopin
Aix Marseille Univ, CNRS, LIS, Marseille, France

Shantanu Das
Aix Marseille Univ, CNRS, LIS, Marseille, France

Maria Kokkou
Aix Marseille Univ, CNRS, LIS, Marseille, France

Abstract

The problem of electing a unique leader is central to all distributed systems, including programmable matter systems where particles have constant size memory. In this paper, we present a silent self-stabilising, deterministic, stationary, election algorithm for particles having constant memory, assuming that the system is simply connected. Our algorithm is elegant and simple, and requires constant memory per particle. We prove that our algorithm always stabilises to a configuration with a unique leader, under a daemon satisfying some fairness guarantees (*Gouda fairness* [27]). We use the special geometric properties of programmable matter in 2D triangular grids to obtain the first self-stabilising algorithm for such systems. This result is surprising since it is known that silent self-stabilising algorithms for election in general distributed networks require $\Omega(\log n)$ bits of memory per node, even for ring topologies [20].

2012 ACM Subject Classification Computer systems organization → Fault-tolerant network topologies; Computing methodologies → Self-organization

Keywords and phrases Leader Election, Programmable Matter, Self-Stabilisation, Silent, Deterministic, Unique Leader, Simply Connected, Gouda fair Daemon, Constant Memory

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.13

Related Version *Full Version*: `https://arxiv.org/abs/2408.08775` [11]

Funding This work has been partially supported by ANR project DUCAT (ANR-20-CE48-0006).

1 Introduction

Leader election (LE), introduced by Le Lann [32], allows to distinguish a unique process in the system as a *leader*. The leader process can then act as an initiator or a coordinator, for solving other distributed problems. Thus, election algorithms are often used as building blocks for many problems in this domain. We are interested in deterministic election algorithms that are *self-stabilising*. Since the seminal work of Dijkstra [18], the self-stabilisation paradigm has been thoroughly investigated (see [19] for a survey). A distributed algorithm is self-stabilising if when executed on a distributed system in an arbitrary global initial configuration, the system eventually reaches a legitimate configuration. Self-stabilising protocols are able to autonomously recover from transient memory failures, without external intervention. A self-stabilising algorithm is *silent* if the system always reaches a configuration where the processes no longer change their states. In the self-stabilising setting, LE is particularly important, as many self-stabilising algorithms rely on the existence of a distinguished node.

The concept of silent self-stabilising algorithms is also related to *proof-labelling* schemes [31] where each node is given a local certificate to verify certain global properties of the system (e.g., the existence of a unique leader). Each node can check its own certificate and those of its neighbours to verify it is in a correct configuration. If the global configuration is incorrect,

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 13; pp. 13:1–13:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

at least one node should be able to detect an inconsistency using the local certificates. In this case, this node will change its state, leading its neighbours to change their states and so on, until the system stabilises to a correct configuration. Blin et al. [5] proved that from any proof-labelling scheme where each process has a certificate of size ℓ, one can build a silent self-stabilising algorithm using $O(\ell + \log n)$ bits of memory per process, where n is the number of processes in the network. One of the standard techniques for self-stabilising LE is to build a spanning tree rooted at the leader, with all other nodes pointing towards their parent in the tree. In order to detect cycles when the system is in an incorrect state, the local certificate at each node includes the hop-distance to the root, in addition to the pointer to the parent. So the size of the certificate depends on the size of the system.

Here, we consider *programmable matter* (PM) systems which are distributed systems consisting of small, intelligent particles that connect to each other and can autonomously change shapes according to input signals. Such systems should be scalable to arbitrary sizes, so the particles have constant size memory independent of the size of the system, similar to finite state automata. This requirement also implies that the particles are anonymous (i.e., do not have unique identifiers) and all communication is limited to $O(1)$ size messages. One well-studied model for PM is the *Amoebot* model [15] where particles operate on a triangular grid (see Section 1.2). LE is a well studied problem in this model. When the system is simply connected, there are stationary deterministic algorithms for election based on the *erosion* approach [17] where the algorithm starts by deactivating particles on the boundary and moving inwards, until the last active node becomes the leader. This approach works under the minimum assumptions on the system and is the inspiration for our work.

Tolerating faults is important for PM, however none of the existing algorithms for election in these systems are self-stabilising. The question is: given the constant memory of particles, is it still possible to obtain a self-stabilising algorithm for PM, using other properties of such systems? We answer this question in the affirmative for *simply connected* PM systems, showing that in this case, a deterministic silent self-stabilising algorithm for LE is indeed possible. We use the property of such systems that there is a unique boundary in the system that is well defined, such that any particle can determine whether it is on the boundary.

1.1 Our results

We present a silent self-stabilising, deterministic, stationary, election algorithm for constant-memory particles, in a simply connected system. We prove our algorithm always stabilises to a unique leader configuration, under a sequential scheduler with some fairness guarantees.

We first present a proof labelling scheme ensuring the existence of a unique leader. Our certificate orients the edges of the network and a configuration is valid when: every edge is oriented, outgoing edges appear consecutively around each particle and there are no directed triangles. Note that our certificate does not ensure that the global orientation of the network is acyclic. However, using the geometric properties of the configuration, we are able to show that any valid configuration has a unique sink. As we are interested in a constant memory algorithm, one cannot transform our proof labelling scheme into a self-stabilising algorithm using [5]. However, we design a very simple algorithm to orient the edges of the system. We show that under our fairness assumption, one always reaches a valid configuration and that this configuration contains a unique sink that is designated as the leader.

Following the classification of [21], our scheduler is Gouda fair [27]: for any configuration C that appears infinitely often in the execution, any successor C' of C also appears infinitely often. Since each particle has constant memory, there exists only a finite number of global configurations of the system. In this setting, Gouda fairness ensures that any configuration that is infinitely often reachable is eventually reached. Observe that a scheduler that at each step activates a particle chosen uniformly at random is a Gouda fair sequential scheduler.

We do not assume that there exists an agreement on the orientation of the grid, or even on its chirality. Observe that without simple connectivity and without agreement on orientation or chirality, it is possible to construct arbitrarily large rings of even size where all processes have the same geometric information about the system (see Figure 1). In this setting, the results of [20] show that there is no silent self-stabilising LE algorithm using constant memory. This is part of our motivation for considering simply connected systems.

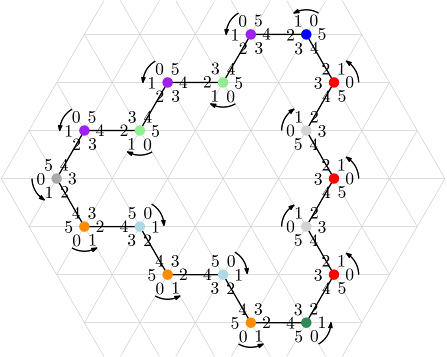


Figure 1 An 18-particle ring where for each particle, the occupied neighbours are reached through port numbers 2 and 4. Two nodes have the same colour if they agree on the grid orientation.

1.2 Related Work

In general networks, there is no self-stabilising leader election algorithm where each process has a constant memory. More precisely, Dolev et al. [20] established that any silent self-stabilising algorithm electing a leader in the class of rings requires $\Omega(\log n)$ bits of memory per process (where n is the size of the ring). This lower bound only uses the assumption that there exists a silent correct configuration and holds for any kind of scheduler. More recently, Blin et al. [4] showed that non-silent self-stabilising algorithms require $\Omega(\log \log n)$ bits of memory per process in order to elect a leader synchronously in the class of rings. Note that these lower bounds are tight in the sense that for rings, there exist silent (resp. non-silent) self-stabilising LE algorithms using $O(\log n)$ (resp., $O(\log \log n)$) bits of memory per process [12, 6]. Constant memory self stabilising algorithms for rings can be designed under special assumptions, as in [30] which gives an algorithm for prime sized rings assuming a sequential scheduler. However, [20] established that this algorithm cannot be made silent.

There exists a large literature about distributed systems where each process has finite memory. Cellular automata, introduced in the 40s in [34] are one of the best known models of this kind. More recently, numerous papers have been devoted to population protocols introduced in [1]. In this model, there is a population of finite-state agents and at each step, a scheduler picks two agents that jointly update their states according to their current states. The scheduler satisfies the same fairness condition as the one we consider in this paper: any configuration that is infinitely often reachable is eventually reached. In this setting, there exist election protocols using only two states when all agents start in the same state. However, when considering self-stabilising LE in this setting, Cai et al. [9] showed that a protocol using $n - 1$ states cannot solve the problem in a population of n agents. This shows that even with a Gouda fair scheduler, it is not always possible to solve the LE problem in a self-stabilising way when processes have constant memory.

PM was introduced in [33] and has since gained popularity. Several models have been introduced, such as [28, 35, 24]. In this paper we consider the well studied *Amoebot* model [15, 13]. In this model, constant-memory computational entities, called *particles*, operate in a triangular grid. Each node of the grid is occupied by at most one particle and particles can determine whether nodes at distance one are occupied by particles. Each particle can communicate with its neighbours by reading their respective registers. It is usually assumed that particles do not have any global sense of direction, while some papers assume that the particles have a common sense of rotational orientation, called *chirality* (e.g., [22, 3]) or that particles agree on a common direction (e.g., [10]). In Amoebot, particles have the ability to move to neighbouring nodes (e.g., [22, 23]), which we do not use. The problem of LE has been studied in the specific context of PM in both 2D (e.g., [3, 17]) and 3D settings (e.g., [26, 8]) and both deterministic (e.g., [22]) and randomized algorithms (e.g., [16]) have been proposed. The existing algorithms for LE in PM can be categorized based on the use of two main techniques: *erosion* (e.g., [17, 25]) and message passing on boundaries (e.g., [3, 16]).

Research on self-stabilisation in the PM setting is more limited. In [16], a randomised LE algorithm is given and the authors discuss the possibility of making it self-stabilising by combining it with techniques from [2, 29]. However, it is assumed that particles have $O(\log^* n)$ memory. In the same paper, it is argued that self-stabilisation in PM is not possible for problems where movement is needed, as the system can become permanently disconnected. A self-stabilising algorithm for constructing a spanning forest was introduced in [14]. The algorithm in [14] is deterministic and particles have constant memory. However, it is assumed that at least one non-faulty special particle always remains in the system. The need to extend the Amoebot model to also address self-stabilising algorithms is also discussed in [13].

2 Model

Let G_Δ be an infinite regular triangular grid where each node has six neighbours. A connected particle system, $\mathcal{P}$, is simply connected if $G_\Delta \backslash \mathcal{P}$ is connected. We assume each node of the simply connected $\mathcal{P}$ contains exactly one particle. We call nodes that are in $\mathcal{P}$ *occupied* and those that are not in $\mathcal{P}$, *empty*. Each particle is anonymous, has constant memory and is stationary (i.e., does not move). A particle is incident to six *ports*, leading to consecutive neighbouring nodes in G_Δ. Each port is associated with a label so that ports i and $i + 1 \mod 6$ lead to neighbouring nodes. A particle knows if each port leads to an occupied or empty node. For each occupied neighbour q, the particle p knows the label assigned by q to qp. Each particle has a constant-size register with arbitrary initial contents. A particle can read the register of each occupied neighbour but can only write in its own register. All particles are *inactive* unless activated by the scheduler. An activated particle reads the contents of its register and the register of each of the neighbouring particles. Based on this information it updates the contents of its own register according to the given algorithm.

We call $\mathcal{P}$, the *support* of the particle system. The configuration C of the system at any time, consists of the set $\mathcal{P}$ and the contents of the registers of each particle in $\mathcal{P}$. A distributed algorithm $\mathcal{A}$ is a set of local rules that particles execute. The rules of the algorithm depend only on the content of the registers of the particles and of its neighbours and they modify only the register of the particle. For an algorithm $\mathcal{A}$, a configuration C, and a particle p, we say that p is *activable* in C, if the execution of $\mathcal{A}$ modifies the contents of the register of p. For two configurations, C and C' that have the same support, we say that C' is a successor of C if there exists an activable particle p in C such that, when p executes $\mathcal{A}$, C' is obtained. An execution $\mathcal{S}$ is an infinite sequence of configurations $\mathcal{S} = C_0, C_1, \ldots$ such that for any i, C_i and C_{i+1} have the same support and either there exists an activable particle p_i such that when p_i executes $\mathcal{A}$ in C_i, C_{i+1} is obtained, or there is no activable particle and $C_{i+1} = C_i$. If there exists a step where $C_{i+1} = C_i$, we call C_i a *final configuration.* An execution is *Gouda fair* [21, 27] if for any configuration C that appears infinitely often in the execution, any successor C' of C also appears infinitely often. An algorithm $\mathcal{A}$ is silent self-stabilising under a *Gouda fair* scheduler, if any such execution of the algorithm contains a final configuration C^* that is valid. The notion of valid configurations depends on the algorithm. In the next section, we define the valid configurations we consider in this paper.

We now present some notations and observations about the geometry of the system. Let v and v' be two neighbouring nodes in $\mathcal{P}$. We say that an edge that is oriented from v to a neighbouring node v' is *outgoing* for v and *incoming* for v'. We write $\overrightarrow{vv'}$ to denote an edge directed from v to v' and vv' to denote an undirected edge or an edge whose orientation is not known. Particles with at least one neighbour that is not in $\mathcal{P}$ are on the *boundary.* Since $\mathcal{P}$ is simply connected, there exists only one boundary in the system. Let p be a particle on the boundary. We say that p is *pending* if p has a unique neighbouring particle in $\mathcal{P}$. We say that p is an *articulation point* if the removal of p disconnects $\mathcal{P}$. If p is neither pending, nor an articulation point, then p is incident to two distinct edges pq, pr on the boundary of $\mathcal{P}$. In this case, since $\mathcal{P}$ is simply connected, there is a path of particles in the 1-neighbourhood of p from q to r. We say that p is on a $\theta \in \{60°, 120°, 180°, 240°\}$ angle to denote the angle that is formed when moving from q to r around p and no empty nodes are encountered. By slight abuse of notation, we also call a particle on a θ angle a θ *particle.* It is easy to see that a particle on the boundary cannot be on a 300° angle, otherwise q and r are adjacent and p is not on the boundary, a contradiction. $\mathcal{P}$ is 2–connected if it does not contain any articulation point. Notice that in systems with at least three particles, a system with no articulation point does not contain any pending particle. In a 2–connected particle system, the following observation implies that there should be a 60° or a 120° particle.

▶ **Observation 1.** *If $\mathcal{P}$ is 2–connected and $|\mathcal{P}| \geq 3$, particles on the boundary satisfy the formula $2n_{60} + n_{120} - n_{240} = 6$, where n_θ is the number of θ particles on the boundary.*

Proof. If $\mathcal{P}$ is 2–connected, it forms a simple polygon. The sum of internal angles of a simple polygon is $(n-2)\pi$, where n is the number of vertices of the polygon. So $(n_{60} + n_{120} + n_{180} + n_{240} - 2)\pi = n_{60}\frac{\pi}{3} + n_{120}\frac{2\pi}{3} + n_{180}\pi + n_{240}\frac{4\pi}{3}$, that is, $2n_{60} + n_{120} - n_{240} = 6$. ◀

▶ **Lemma 2.** *In any simply connected particle system $\mathcal{P}$ with at least two particles, the boundary of $\mathcal{P}$ contains one of following:*

1. *a pending particle, or*
2. *a 60° particle, or*
3. *two 120° particles that are connected by a path of 180° particles on the boundary.*

Proof. A block is a 2–connected component of $\mathcal{P}$. As $\mathcal{P}$ contains at least two particles, each block is either an edge or it contains at least three particles. The block tree of $\mathcal{P}$ is a tree where each vertex is a block and there is an edge between two blocks if they share a vertex (i.e., an articulation point of $\mathcal{P}$). A leaf, $\mathcal{P}'$, of the block tree is a 2–connected component of $\mathcal{P}$ and contains a unique articulation point p' of $\mathcal{P}$. If $\mathcal{P}'$ contains precisely two particles p' and q', then p' is the unique neighbour of q' in $\mathcal{P}$ and q' is a pending particle, as in Case 1.

Suppose $\mathcal{P}'$ contains at least three particles. Since $\mathcal{P}'$ is 2–connected, every particle on the boundary of $\mathcal{P}'$ is a $\theta \in \{60°, 120°, 180°, 240°\}$ particle. Any θ particle $p \neq p'$ of $\mathcal{P}'$ is also a θ particle of $\mathcal{P}$. So a 60° particle $p \neq p'$ in $\mathcal{P}'$, is a 60° particle in $\mathcal{P}$, which is Case 2.

Suppose now that in $\mathcal{P}'$, any boundary particle p different from p' is a θ particle with $\theta \in \{120°, 180°, 240°\}$. Let n'_{120}, n'_{180}, n'_{240} be respectively the number of 120°, 180°, 240° particles in $\mathcal{P}'$ that are different from p'. Since p' is an articulation point, p' cannot have more than three consecutive particle neighbours. Consequently, in $\mathcal{P}'$, p' is either a 60° or a 120° particle. If p' is a 60° particle in $\mathcal{P}'$, from Observation 1, we have $2 + n'_{120} - n'_{240} = 6$. If p' is a 120° particle in $\mathcal{P}'$, from Observation 1, we have $n'_{120} + 1 - n'_{240} = 6$. In both cases, we then have $n'_{120} \geq n'_{240} + 4$. Let $p_1, \ldots, p_{n'_{120}}$ be the 120° particles of $\mathcal{P}'$ in the order in which they appear when we move on the boundary of $\mathcal{P}'$ starting from p' (i.e., p' appears between $p_{n'_{120}}$ and p_1). Since $n'_{120} \geq n'_{240} + 4 > n'_{240} + 1$, there exists an index $1 \leq i \leq n'_{120} - 1$ such that only 180° particles appear on the boundary of $\mathcal{P}'$ between p_i and p_{i+1}. Since all these 180° particles are also 180° particles on the boundary of $\mathcal{P}$, we are in Case 3. ◀

We explain how two adjacent particles in a triangle detect each other's chirality. The label $\lambda(\Pi)$ of a path $\Pi = (p_1, p_2, \ldots, p_k)$ in the graph induced by the particles is a sequence of pairs of labels $(\mathsf{a}_1, \mathsf{b}_2), (\mathsf{a}_2, \mathsf{b}_3), \ldots, (\mathsf{a}_{k-1}, \mathsf{b}_k)$ where for each i, a_i (resp. b_i) is the port connecting p_i to p_{i+1} (resp. p_{i-1}).

Following [36], we define the *view* of depth k of a particle p, denoted by $\mathtt{view}_k(p)$, to be the set of labels $\lambda(\Pi)$ of paths Π starting at p of length at most k. Note that for each $1 \leq j \leq k$, if both $(\mathsf{a}_1, \mathsf{b}_2), (\mathsf{a}_2, \mathsf{b}_3), \ldots, (\mathsf{a}_j, \mathsf{b}_{j+1})$ and $(\mathsf{a}_1, \mathsf{b}_2), (\mathsf{a}_2, \mathsf{b}_3), \ldots, (\mathsf{a}_j, \mathsf{b}'_{j+1})$ belong to $\mathtt{view}_k(p)$, then $\mathsf{b}_{j+1} = \mathsf{b}'_{j+1}$. From [7], for any constant k, each particle p can construct $\mathtt{view}_k(p)$ in a self stabilising way with constant memory.

▶ **Lemma 3.** *For any triangle of particles pqr, p can infer the chirality of q from* $\mathrm{view}_3(p)$*.*

Proof. In the following, for a particle p, we let $\{\mathsf{p}_i \mid 0 \leq i \leq 5\}$ be the set of ports incident to p and we assume that either $\mathsf{p}_{i+1} = \mathsf{p}_i + 1$ for each $0 \leq i \leq 5$, or $\mathsf{p}_{i+1} = \mathsf{p}_i - 1$ for each $0 \leq i \leq 5$ (where additions are made modulo 6). We will use the following observation.

▷ Claim 4. If pqr is a triangle, then the ports connecting r to p and q are consecutive.

Consider a triangle pqr. Let p_1 (resp. q_1) be the port through which p (resp. q) is connected to q (resp. p). Further, let p (resp. r) be connected to r (resp. p) through p_0 (resp. r_1). Observe that if p learns the port through which q is connected to r, it also learns the chirality of q. Note that by Claim 4, this port is either q_0 or q_2 and the port from r to q is either r_0 or r_2. Notice that if r is the only common neighbour of p and q, then only one of $\{(\mathsf{p}_1, \mathsf{q}_1), (\mathsf{q}_0, \mathsf{x}) \mid 0 \leq \mathsf{x} \leq 5\} \cup \{(\mathsf{p}_1, \mathsf{q}_1), (\mathsf{q}_2, \mathsf{x}) \mid 0 \leq \mathsf{x} \leq 5\}$ is in $\mathtt{view}_3(p)$ and p can then infer the chirality of q. Suppose now that p and q have two common neighbours r and r'.

▷ Claim 5. The edge qr is labelled $(\mathsf{q}_2, \mathsf{r}_0)$ if and only if the following formula holds:

$$\begin{aligned}(\mathsf{p}_1, \mathsf{q}_1), (\mathsf{q}_2, \mathsf{r}_0), (\mathsf{r}_1, \mathsf{p}_0) \in \mathtt{view}_3(p) \;\wedge\; (\mathsf{p}_0, \mathsf{r}_1), (\mathsf{r}_0, \mathsf{q}_2), (\mathsf{q}_1, \mathsf{p}_1) \in \mathtt{view}_3(p) \\ \wedge\; \Big[(\mathsf{p}_1, \mathsf{q}_1), (\mathsf{q}_0, \mathsf{r}_2) \notin \mathtt{view}_3(p) \;\vee\; (\mathsf{p}_2, \mathsf{r}_5) \notin \mathtt{view}_3(p) \Big]\end{aligned}$$

Proof. First let us suppose that the edge qr is labelled $(\mathsf{q}_2, \mathsf{r}_0)$. Then, the first two expressions of the formula are satisfied. Let us suppose $(\mathsf{p}_1, \mathsf{q}_1), (\mathsf{q}_0, \mathsf{r}_2) \in \mathtt{view}_3(p)$. Then from Claim 4, qr' is labelled $(\mathsf{q}_0, \mathsf{r}_2)$ and pr' is either labelled $(\mathsf{p}_2, \mathsf{r}_1)$ or $(\mathsf{p}_2, \mathsf{r}_3)$. In either case, $(\mathsf{p}_2, \mathsf{r}_5) \notin \mathtt{view}_3(p)$ and the formula is satisfied.

Let us now suppose that the formula is satisfied and assume that qr is not labelled $(\mathsf{q}_2, \mathsf{r}_0)$. Then by Claim 4, qr is labelled either $(\mathsf{q}_2, \mathsf{r}_2)$, or $(\mathsf{q}_0, \mathsf{r}_0)$, or $(\mathsf{q}_0, \mathsf{r}_2)$. Note that the first two cases are impossible since $(\mathsf{p}_1, \mathsf{q}_1), (\mathsf{q}_2, \mathsf{r}_0)$ and $(\mathsf{p}_0, \mathsf{r}_1), (\mathsf{r}_0, \mathsf{q}_2)$ belong to $\mathtt{view}_3(p)$. Consequently, qr is labelled $(\mathsf{q}_0, \mathsf{r}_2)$ and since $(\mathsf{p}_1, \mathsf{q}_1)(\mathsf{q}_2, \mathsf{r}_0) \in \mathtt{view}_3(p)$, by Claim 4, qr' is labelled $(\mathsf{q}_2, \mathsf{r}_0)$, and the label of pr' is either $(\mathsf{p}_2, \mathsf{r}_5)$ or $(\mathsf{p}_2, \mathsf{r}_1)$. Note that we are necessarily in the second case since we assumed that the formula holds and since $(\mathsf{p}_1, \mathsf{q}_1)(\mathsf{q}_0, \mathsf{r}_2) \in \mathtt{view}_3(p)$. This implies that $(\mathsf{p}_1, \mathsf{q}_1)(\mathsf{q}_2, \mathsf{r}_0)(\mathsf{r}_1, \mathsf{p}_2) \in \mathtt{view}_3(p)$, and thus $(\mathsf{p}_1, \mathsf{q}_1)(\mathsf{q}_2, \mathsf{r}_0)(\mathsf{r}_1, \mathsf{p}_0) \notin \mathtt{view}_3(p)$, contradicting the fact that the formula holds. ◁

From Claim 4, qr must be labelled $(\mathsf{q}_0, \mathsf{r}_0)$, $(\mathsf{q}_0, \mathsf{r}_2)$, $(\mathsf{q}_2, \mathsf{r}_0)$ or $(\mathsf{q}_2, \mathsf{r}_2)$. Applying Claim 5 to each possibility, p can detect the label of qr and thus infer the chirality of q. ◀

3 A Proof Labelling Scheme for Leader Election

Our aim is to orient all edges so that a unique sink particle (i.e., particle with no outgoing edges) that we define to be the leader exists. The certificate given to each particle consists of a direction for each edge incident to the particle. The orientation of the edges is chosen so that particles that are reached by an outgoing edge of some particle p, induce a connected graph of size at most three. In general we cannot avoid the existence of cycles in the orientation, but we will show that the existence of a unique sink is always guaranteed. Each particle p checks that the following rules are locally satisfied or detects an error.

R1 Each edge is oriented and both particles agree on the direction of the edge.
R2 Particle p has at most three outgoing edges. We consider edges between p and empty nodes to be incoming for p.
R3 When looking at the ports of p cyclically, all outgoing edges of p are consecutive.
R4 For every 3-particle triangle p belongs in, the triangle is not a cycle.

We call a configuration where every particle satisfies R1–R4 a *valid configuration*. Note that R4 does not guarantee an acyclic orientation (i.e., that larger cycles do not exist in the configuration). We do not forbid global cycles, but we will prove that even if cycles of size larger than three are formed by the incoming and outgoing edges, the remaining rules guarantee that there exists a unique sink in the system that we define to be the leader.

▶ **Theorem 6.** *If all rules R1–R4 are satisfied, then there exists a unique sink in the system.*

A valid configuration that does not contain any oriented cycle is a *valid acyclic orientation*. Observe that any particle system admits a valid acyclic orientation, as it can be constructed from any erosion based Leader Election algorithms for programmable matter (e.g., [17, 22]). Indeed, consider an execution of an erosion algorithm on a system, and orient any edge pq from p to q if p is eroded before q in the execution. This orientation is acyclic and it thus obviously satisfies R1 and R4. Since an erosion based algorithm erodes only a particle that does not disconnect its neighbourhood and that is strictly convex (i.e., that has at most three non-eroded neighbours), the orientation also satisfies R2 and R3.

4 A Self-Stabilising Algorithm for Leader Election

In Section 3, we claimed that *if* all rules are locally satisfied a unique sink exists in the system. Here, we show *how* a valid configuration is reached from a configuration containing errors. Our algorithm is simple: when a particle, p, is incident to an undirected edge e, p orients e as outgoing. If the orientation of the edges incident to p violates a rule, p undirects all its outgoing edges. Each activated particle always executes both lines of Algorithm 1.

Algorithm 1 Self Stabilising LE.

```
If  ¬R1              :  Mark all undirected edges as outgoing
If  ¬R2 ∨ ¬R3 ∨ ¬R4  :  Mark all outgoing edges as undirected
```

The directed edges incident to particles are encoded by each particle p having a variable $link_p[v'] \in \{in, out\}$ for each neighbouring node v'. For particle $p \in \mathcal{P}$ and $v' \in V_{G_\Delta \setminus \mathcal{P}}$, $link_p[v'] = in$. Any particle p can locally detect whether it is incident to an empty node, so we assume that edges between occupied and unoccupied nodes are always marked correctly. For two adjacent particles $p, p' \in \mathcal{P}$, if $link_p[p'] = in$ and $link_{p'}[p] = out$, pp' is directed from p' to p. We encode an *undirected* edge between two particles p and p' as $link_p[p'] = link_{p'}[p] = in$. We address $link_p[p'] = link_{p'}[p] = out$, as a special case. The endpoint that is activated first (say p) marks $link_p[p']$ as in. Notice that this is only possible during the first activation of p. In the remainder of this paper we only use the orientation of the edges without referencing their encoding. That is, we say that an edge between two particles p and q is: directed from p to q (i.e., $\overrightarrow{pq}$), directed from q to p (i.e., $\overrightarrow{qp}$) or undirected (i.e., pq or qp). From Lemma 3 particles in a common triangle detect each other's chirality. Since particles know both labels assigned to an edge, particles can compute the orientation of edges in triangles they belong in and check R4. From now on we only refer to particles detecting cyclic triangles. We prove that when executing our algorithm, any particle system reaches a valid configuration that contains a unique sink.

▶ **Theorem 7.** *Starting from an arbitrary simply connected configuration any Gouda fair execution of Algorithm 1 eventually reaches a configuration satisfying R1–R4 in which no rules can be applied and there exists a unique sink.*

5 Proof of Theorem 6 and Theorem 7

Here, we prove Theorem 7. Notice the second statement of Theorem 7 is precisely Theorem 6. A configuration of a particle system executing Algorithm 1 is described by the direction of each edge pq (i.e., $\overrightarrow{pq}$, $\overrightarrow{qp}$ or undirected). We make a few observations on how to change the orientation of some edges of a valid configuration and maintain a valid configuration.

▶ **Observation 8.** *Let p be a particle such that R3 is satisfied at p. Let e be an incoming edge to p and e' be an outgoing edge of p, s.t. when moving cyclically around p, e and e' are consecutive. If e becomes outgoing (resp. e' becomes incoming), R3 is not violated at p.*

▶ **Observation 9.** *Let C be a configuration and let p be a particle so that R1 (resp., R2, R4) is satisfied at some particle $q \neq p$ in C. Then, R1 (resp., R2, R4) is satisfied at q in $C \setminus \{p\}$.*

Let $\mathcal{S} = C_0, C_1, \ldots$ be an execution of Algorithm 1 starting from a configuration C_0. Notice a particle p is activable in a configuration C if when it executes Algorithm 1, one of its undirected edges becomes outgoing or one of its outgoing edges becomes undirected. If there exists a configuration C_f where no node is activable, then $C_f = C_j$ for all $j > f$, and we say that the execution stabilises to a *final* configuration. If all rules are satisfied in this final configuration, then this configuration is valid and we say it is a *final directed* configuration. If a configuration C_i is not final, we can assume that there exists an activable particle p_i such that we obtain C_{i+1} by activating p_i in C_i. Since each particle has constant memory, the number of possible configurations is finite. Hence there exists an index i_0 in $\mathcal{S}$ such that any configuration C_i with $i \geq i_0$ appears infinitely often in $\mathcal{S}$. We write $\mathcal{S}_{i_0} = C_{i_0}, C_{i_0+1}, \ldots$

to denote the part of the execution starting at C_{i_0} and in the following we consider only $\mathcal{S}_{i_0}$ and configurations C_i with $i \geq i_0$. We call the edges that are never undirected in $\mathcal{S}_{i_0}$, *stable edges*. Observe that by the definition of i_0, each edge is either stable or undirected infinitely often. Notice that any edge pq directed from p to q in C_i with $i \geq i_0$, is directed from p to q infinitely often, regardless of whether it is stable. We establish some properties in $\mathcal{S}_{i_0}$.

▶ **Lemma 10.** *In any configuration C_i with $i \geq i_0$, Rules R2, R3 and R4 are always satisfied.*

Proof. Let n_i be the number of particles in C_i that do not satisfy R2, R3 or R4. If a particle p is activated in C_i, then R2, R3 and R4 are satisfied at p in C_{i+1}. Moreover, if R2, R3, and R4 are satisfied at some particle p in C_i that is not activated at step i, then they are still satisfied at step C_{i+1}. Consequently, $n_{i+1} \leq n_i$. Since for $i \geq i_0$, C_i appears infinitely often, we get that for every $i \geq i_0$, we have $n_i = n_{i_0}$. If $n_i > 0$, there exists a particle p that always violates one of the rule R2, R3 or R4. Thus, p is eventually activated at some step i and in C_{i+1}, p satisfies the rules, a contradiction. Consequently, for any $i \geq i_0$, R2, R3 and R4 are satisfied at every particle in C_i. ◀

▶ **Lemma 11.** *If a particle p is incident to a stable outgoing edge, p is never activable in $\mathcal{S}_{i_0}$ and all edges incident to p are stable edges.*

Proof. In a configuration C_i, if a particle p is incident to an outgoing edge and an undirected edge, then p is activable in C_i. After its activation, either all the undirected edges incident to p have become outgoing edges, or all outgoing edges of p have become undirected.

Let $\overrightarrow{pq}$ be a stable edge, hence, p never marks $\overrightarrow{pq}$ as undirected. Let us suppose that in addition to $\overrightarrow{pq}$, p is also incident to an unstable edge pr. Then infinitely often, pr is undirected and thus there exists a step where p is activated and pr is undirected. At this step, p marks $\overrightarrow{pr}$ as outgoing. Then if $\overrightarrow{pr}$ becomes undirected at a later step, $\overrightarrow{pq}$ must also become undirected, which is a contradiction. Hence, all edges incident to p are stable. ◀

▶ **Lemma 12.** *If a particle p is incident to an unstable edge in $\mathcal{S}_{i_0}$, the unstable edges incident to p are at least two and do not appear consecutively around p, or there are at least four unstable edges incident to p.*

Proof. Suppose the lemma does not hold and that there exists a particle p incident to $1 \leq k \leq 3$ unstable edges that appear consecutively around p. Then, there exists an unstable edge pq incident to p such that for every unstable edge pr incident to p, either $r = q$ or r is adjacent to q. Note that by Lemma 11, all stable edges incident to p are incoming to p.

▷ Claim 13. Consider an unstable edge pr with $r \neq q$ and let s be the common neighbour of p and r that is distinct from q. Then $\overrightarrow{sp}$ and $\overrightarrow{sr}$ are stable.

Proof. By the definition of q, sp is stable, and by Lemma 11, sp is oriented from s to p. By Lemma 11 applied at s and r, sr is also stable and it is directed from s to r. ◁

Suppose first there is a configuration C_i with $i \geq i_0$ such that pq is directed from p to q in C_i and undirected in C_{i+1}. This implies that p is activated at step i. By Lemma 10 there exists at least an undirected edge pr in C_i, and when orienting all undirected edges incident to p in C_i as outgoing edges, one of R2, R3, R4 is violated. By the definition of q, this cannot be R2 or R3. If R4 is violated, it implies that in C_i, there exists an undirected edge pr and directed edges $\overrightarrow{rs}$ and $\overrightarrow{sp}$. By Claim 13, $s = q$ but this is impossible since $\overrightarrow{pq}$ is in C_i.

Then, at each step $i \geq i_0$, either qp is undirected or it is $\overrightarrow{qp}$. If there is no step $i \geq i_0$ where q is activated, then q never has any outgoing edge, qp is always undirected and we let $i_1 = i_0$. Otherwise, consider a step $i_1 - 1$ where q is activated such that in C_{i_1}, qp is undirected. Then at step i_1, all edges incident to p are either incoming or undirected. We claim that if we activate p at step i_1, it orients pq from p to q. Indeed by the definition of q, rules R2 and R3 are satisfied when pq is oriented from p to q. By Claim 13, any triangle violating R4 should contain q, but this is impossible since q has no outgoing edges in C_{i_1}. So, by the fairness condition, there exists a configuration C_i containing $\overrightarrow{pq}$, a contradiction. ◀

We now prove Theorem 7 using the structure of the boundary of $\mathcal{P}$ given by Lemma 2. Informally, the proof has the following structure. We assume that it is possible that the system does not stabilise and arrive at a contradiction. Out of the particle systems that do not stabilise to a configuration that satisfies all rules and has a unique sink, we take a system with the minimum number of particles. On the boundary of that system there exists a particle p satisfying one of the cases of Lemma 2. For each orientation of the edges incident to p we show that the edges incident to p are stable. Then we take a smaller system containing exactly one less particle, p. We show that the execution in both systems for particles that are not p is the same. Hence, if the system that contains p does not satisfy all rules and does not have a unique sink, the same is true for the system that does not contain p. Since we had assumed that the system containing p is the minimum size system that does not stabilise to a valid configuration, a smaller system not stabilising is a contradiction.

Proof of Theorem 7. Let us suppose that there exists a fair execution $\mathcal{S} = C_0, C_1, \ldots$ on a particle configuration $C = C_0$ that does not stabilise to a final directed configuration containing a unique sink. Consider such an execution $\mathcal{S}$ with a support $\mathcal{P}$ of minimum size. As defined above, consider a fair execution $\mathcal{S}_{i_0} = C_{i_0}, C_{i_0+1}, \ldots$ containing only configurations appearing infinitely often. By Lemma 2, we can assume that the boundary of $\mathcal{P}$ contains either a pending particle, or a 60° particle, or two 120° particles that are connected by a path of 180° particles on the boundary. In the following, we show that each of these cases cannot occur. We first consider the case where $\mathcal{P}$ contains a pending particle.

▶ **Lemma 14.** *If $\mathcal{P}$ contains a pending particle p (i.e., a particle with only one neighbouring particle w), all edges are stable in $\mathcal{S}_{i_0}$ and there is a unique sink in the final configuration.*

Proof. By Lemma 12, the edge pw is stable. Suppose first that pw is directed from p to w in $\mathcal{S}_{i_0}$. For each $i \geq i_0$, let $C'_i = C_i \setminus \{p\}$ and consider the sequence of configurations $\mathcal{S}'_{i_0} = C'_{i_0}, C'_{i_0+1}, \ldots, C'_i, \ldots$. Observe that for each $i \geq i_0$, either $C_{i+1} = C_i$ or there exists p_i such that C_{i+1} is obtained from C_i by activating p_i and thus modifying the orientations of edges incident to p_i. Since $\overrightarrow{pw}$ is stable, by Lemma 11, for any $i \geq i_0$, $p_i \neq p$. Moreover, for each $i \geq i_0$, the edges of C'_i have the same orientation as in C_i. So, $p' \neq p$ is activable in C'_i if and only if it is activable in C_i. Furthermore, the configuration obtained by activating p_i in C'_i is precisely C'_{i+1} since the edges of C'_i have the same orientation as in C_i. Hence, $\mathcal{S}'_{i_0}$ is a fair execution of Algorithm 1 on $\mathcal{P} \setminus \{p\}$. By the minimality of the size of $\mathcal{P}$, there exists a step $i_1 \geq i_0$ such that C'_{i_1} is a final directed configuration that contains a unique sink p''. Since the edges incident to p are stable, C_{i_1} is a final directed configuration. By our definition of i_0, this implies that $i_1 = i_0$. Since p has an outgoing edge, $\overrightarrow{pw}$, in $C_{i_1} = C_{i_0}$, p is not a sink of C_{i_0} and p'' is the unique sink in C_{i_0}.

Suppose now that $\overrightarrow{wp}$ is stable. Notice that in this case, p is a sink in C_i for each $i \geq i_0$. Moreover, since $\overrightarrow{wp}$ is stable, by Lemma 11, w is never activated. Since the two common neighbours of p and w are empty, by R3, p is the only outgoing neighbor of w in C_i for any $i \geq i_0$. Consequently, w is a sink in $C_i \setminus \{p\}$ for any $i \geq i_0$. For each $i \geq i_0$, let $C'_i = C_i \setminus \{p\}$ and consider the sequence of configurations $\mathcal{S}'_{i_0} = C'_{i_0}, C'_{i_0+1}, \ldots, C'_i, \ldots$ Observe that for each $i \geq i_0$, either $C_{i+1} = C_i$ or there exists p_i such that C_{i+1} is obtained from C_i by activating p_i and thus modifying the orientations of edges incident to p_i. Since p has only incoming edges in C_i, $p_i \neq p$. Moreover, since $\overrightarrow{wp}$ is stable, $p_i \neq w$. Moreover, for each $i \geq i_0$, the edges of C'_i have the same orientation as in C_i. Consequently, $p' \neq p$ is activable in C'_i if and only if it is activable in C_i. Furthermore, the configuration obtained by activating p_i in C'_i is precisely C'_{i+1} since the edges of C'_i have the same orientation as in C_i. Consequently, $\mathcal{S}'_{i_0}$ is a fair execution of Algorithm 1 on $\mathcal{P} \setminus \{p\}$. By the minimality of the size of $\mathcal{P}$, there exists a step $i_1 \geq i_0$ such that C'_{i_1} is a final directed configuration that contains a unique sink $p'' = w$ in C'_{i_1}. Since $\overrightarrow{wp}$ is stable, C_{i_1} is a final directed configuration. By our definition of i_0, this implies that $i_1 = i_0$. Since any $p' \in C_{i_0} \setminus \{p, w\}$ is not a sink in C'_{i_0}, and since $\overrightarrow{wp}$ is in C_{i_0}, p is the unique sink in the valid configuration C_{i_0}. ◀

We now consider the case where the boundary of $\mathcal{P}$ contains a 60° particle p, and we let q and r be the two neighbours of p on the boundary of $\mathcal{P}$.

▶ **Lemma 15.** *If $\mathcal{P}$ contains a 60° particle p, all edges are stable and there is a unique sink in the final configuration.*

Proof. By Lemma 12, pq and pr are stable. Consequently, pq and pr are always directed in the same way all along $\mathcal{S}_{i_0}$ and we can talk about the orientation of pq and pr in $\mathcal{S}_{i_0}$. For each $i \geq i_0$, let $C'_i = C_i \setminus \{p\}$ and consider the sequence of configurations $\mathcal{S}'_{i_0} = C'_{i_0}, C'_{i_0+1}, \ldots, C'_i, \ldots$. For each $i \geq i_0$, either $C_{i+1} = C_i$ or there exists p_i such that C_{i+1} is obtained from C_i by activating p_i and thus modifying the orientations of edges incident to p_i. Since all edges incident to p are stable in $\mathcal{S}_{i_0}$, we can assume $p_i \neq p$, for any $i \geq i_0$.

We distinguish three cases, depending on the orientation of pq and pr in $\mathcal{S}_{i_0}$.

▷ Case 1. The edges incident to p are $\overrightarrow{pq}$ and $\overrightarrow{pr}$.

Proof. For each $i \geq i_0$, the edges of C'_i have the same orientation as in C_i. Hence, a particle $p' \neq p$ is activable in C'_i if and only if it is activable in C_i. The configuration obtained by activating p_i in C'_i is precisely C'_{i+1} since the edges of C'_i have the same orientation as in C_i. So, $\mathcal{S}'_{i_0}$ is a fair execution of Algorithm 1 on $\mathcal{P} \setminus \{p\}$. By the minimality of the size of $\mathcal{P}$, there exists a step $i_1 \geq i_0$ such that C'_{i_1} is a final directed configuration with a unique sink p''. Since the edges incident to p are stable, C_{i_1} is a final directed configuration. By our definition of i_0, this implies that $i_1 = i_0$. Since p has only outgoing edges in $C_{i_1} = C_{i_0}$, p is not a sink of C_{i_0} and p'' is the unique sink in C_{i_0}. ◁

▷ Case 2. The edges incident to p are $\overrightarrow{qp}$ and $\overrightarrow{pr}$.

Proof. Since $\overrightarrow{qp}$ is stable, qr is also stable by Lemma 11. By R4, qr is directed from q to r in $\mathcal{S}_{i_0}$. Notice that since q and p are incident to outgoing stable edges, from Lemma 11, p and q are incident only to stable edges and are never activable. The edges of $C'_{i \geq i_0} \setminus \{p\}$ have the same orientation as in $C_{i \geq i_0}$. Consequently, for any $p' \notin \{p, q\}$, p' is activable in C'_i if and only if it is activable in C_i. Let us consider q. In C_i, $\overrightarrow{qp}$ is stable and directed and in C'_i, q has an incoming edge from the respective empty node. Furthermore, q is incident to an incoming edge from the empty common neighbour of p and q. Hence, R3 is satisfied for q in C'_i from Observation 8, and the remaining rules are satisfied for q in C'_i from Observation 9. Therefore, q is never activable in $C'_{i \geq i_0}$ and thus a particle $p' \neq p$ is activable in $C'_{i \geq i_0}$ if and only if it is activable in $C_{i \geq i_0}$. So, $\mathcal{S}'_{i_0}$ is a fair execution of Algorithm 1 on $\mathcal{P} \setminus \{p\}$. By the minimality of the size of $\mathcal{P}$, there exists a step $i_1 \geq i_0$ such that C'_{i_1} is a final directed configuration that contains a unique sink p''. Note that $p'' \neq q$ since $\overrightarrow{qr} \in C'_{i_1}$. Since the edges incident to p are stable and since q is not activable, C_{i_1} is a final directed configuration. By our definition of i_0, this implies that $i_1 = i_0$. Since p has an outgoing edge in $C_{i_1} = C_{i_0}$, p is not a sink of C_{i_0} and p'' is therefore the unique sink in C_{i_0}. ◁

▷ Case 3. The edges incident to p are $\overrightarrow{qp}$ and $\overrightarrow{rp}$.

Proof. Since $\overrightarrow{qp}$ and $\overrightarrow{rp}$ are stable, from Lemma 11, q and r are never activable and all edges incident to q and r are stable. Hence qr is stable and we assume without loss of generality that qr is directed as $\overrightarrow{qr}$ in $C_{i\geq i_0}$. Notice that p is a sink in C and that from R3 all edges incident to r except $\overrightarrow{rp}$ are incoming to r in C_i. Edges in $C'_i \setminus \{p\}$ have the same orientation as in C_i. Using the arguments from Case 2, R1–R4 are satisfied for q and for r in C'_i. So, q and r are never activable in C'_i. For the reasons in Case 2, $\mathcal{S}'_{i_0}$ is a fair execution of Algorithm 1 on $\mathcal{P} \setminus \{p\}$. Since $\mathcal{P}$ is of minimum size, there exists a step $i_1 \geq i_0$ such that C'_{i_1} is a final directed configuration that contains a unique sink p''. Since the only outgoing edge of r in C_{i_0} is $\overrightarrow{rp}$, $r = p''$ is the unique sink of C'_{i_1}. Furthermore, since the edges incident to p, q, r are stable, C_{i_1} is a final directed configuration. By our definition of i_0, this implies that $i_1 = i_0$. Since p has only incoming edges in $C_{i_1} = C_{i_0}$, p is a sink in C_{i_0} and r is not a sink in C_{i_0} due to $\overrightarrow{rp}$, so p is the unique sink in C_{i_0}. ◁

Therefore, for any orientation of the edges incident to p in $\mathcal{S}_{i_0}$, C_{i_0} is a directed final configuration containing a unique sink. ◀

Now assume there exist two 120° particles connected by a path of 180° particles on the boundary of $\mathcal{P}$.

▶ **Lemma 16.** *If $\mathcal{P}$ contains two 120° particle p, p^* connected by a path of 180° particles on the boundary, all edges are stable and there is a unique sink in the final configuration.*

Proof. Let q and r be the neighbours of p on the boundary and let s be the common neighbour of p, q and r. By Lemma 12, we know that ps is stable in $\mathcal{S}_{i_0}$. We split the proof of the lemma in different cases, depending on the orientation of ps in $C_{i\geq i_0}$. Due to space constraints, we omit some details, which can be found in [11], in the proofs of Cases 1.1 – 2.3.

▷ Case 1. The edge between p and s is $\overrightarrow{ps}$.

For each $i \geq i_0$, let $C'_i = C_i \setminus \{p\}$ and consider the sequence of configurations $\mathcal{S}'_{i_0} = C'_{i_0}, C'_{i_0+1}, \ldots, C'_i, \ldots$ For each $i \geq i_0$, either $C_{i+1} = C_i$ or there exists p_i such that C_{i+1} is obtained from C_i by activating p_i and thus modifying the orientations of edges incident to p_i. From Lemma 11, since $\overrightarrow{ps}$ is stable, p is never activable and the edges pq and pr are stable. Consequently, $p_i \neq p$ for any $i \geq i_0$. The orientations of pq and pr lead to the following cases.

▷ Case 1.1. Particle p is incident to $\overrightarrow{pq}$, $\overrightarrow{ps}$ and $\overrightarrow{pr}$.

Proof. The proof for this case follows the same argumentation as Case 1 of Lemma 15. ◁

▷ Case 1.2. Particle p is incident to $\overrightarrow{pq}$, $\overrightarrow{ps}$ and $\overrightarrow{rp}$.

Proof. Since $\overrightarrow{rp}$ is stable, from Lemma 11, r is not activable in C and rs is stable. From R4, $\overrightarrow{rs}$ is in $C_{i\geq i_0}$. The edges of $C'_{i\geq i_0} \setminus \{p\}$ have the same orientation as in $C_{i\geq i_0}$. So for any $p' \notin \{p, r\}$, p' is activable in C'_i if and only if p' is activable in C_i. Let us consider r. From Observation 8 and the incoming edge from the empty common neighbour of p and r to r, R3 is satisfied for r in C'_i. R1, R2 and R4 are satisfied for r in C'_i from Observation 9. Hence, r is never activable in C'_i. So, $\mathcal{S}'_{i_0}$ is a fair execution of Algorithm 1 on $\mathcal{P} \setminus \{p\}$. The unique sink in C'_{i_0} is $p'' \neq r$, since r has an outgoing edge $\overrightarrow{rs}$ in C'_i. Since p has outgoing edges in C_{i_0}, p is not a sink of C_{i_0} and p'' is the unique sink in C_{i_0}. ◁

▷ Case 1.3. Particle p is incident to $\overrightarrow{qp}$, $\overrightarrow{ps}$ and $\overrightarrow{rp}$.

Proof. Since $\overrightarrow{qp}$ and $\overrightarrow{rp}$ are stable, from Lemma 11, r and q are not activable in C and the edges rs and qs are stable. By R4, $\overrightarrow{rs}$ and $\overrightarrow{qs}$ belong to $C_{i \geq i_0}$. The edges of $C'_{i \geq i_0} \setminus \{p\}$ have the same orientation as in $C_{i \geq i_0}$. Consequently, for any $p' \notin \{p, q, r\}$, p' is activable in C'_i if and only if p' is activable in C_i. Let us consider q and r. Using the same arguments as in Case 1.2, R1–R4 are satisfied for q and for r in C'_i. Consequently, q and r are never activable in C'_i. Therefore, C'_i satisfies all rules and $\mathcal{S}'_{i_0}$ is a fair execution of Algorithm 1 on $\mathcal{P} \setminus \{p\}$. Since q and r each have an outgoing edge to s in C'_i, $p'' \notin \{q, r\}$. Since p has outgoing edges in C_{i_0}, p is not a sink of C_{i_0} and p'' is therefore the unique sink in C_{i_0}. ◁

We now consider the case where the edge between p and s is $\overrightarrow{sp}$. Observe that by Lemma 11, sq and sr are stable edges. Note that by Lemma 12, pq is stable if and only if pr is stable. We distinguish two cases depending on whether these two edges are stable.

▷ Case 2. The edge between p and s is $\overrightarrow{sp}$, and the edges pq and pr are stable.

The possible orientations of the stable edges sq and sr in $\mathcal{S}_{i_0}$ give the following subcases.

▷ Case 2.1. Particle s is incident to $\overrightarrow{qs}$ and $\overrightarrow{rs}$.

Proof. By Lemma 11, r and q are never activable and rp and qp are stable. By R4, $\overrightarrow{qp}$ and $\overrightarrow{rp}$ are in $C_{i \geq i_0}$. The edges of $C'_{i \geq i_0} \setminus \{p\}$ have the same orientation as in $C_{i \geq i_0}$ and thus for any $p' \notin \{q, r, s\}$, p' is activable in C'_i if and only if it is activable in C_i. Let us consider q, r, s. Using the same arguments as in Case 1.2, R1–R4 are satisfied for q and r in $C \setminus \{p\}$. Since sp is between two incoming edges, from Observation 8, R3 is satisfied for s in C'_i. R1, R2 and R4 are satisfied for s in C'_i from Observation 9. So, q, r, s are never activable in C'_i. Furthermore, the configuration obtained by activating $p_i \neq q, s, r$ in C'_i is precisely C'_{i+1} since the edges of C'_i have the same orientation as in C_i. Hence, $\mathcal{S}'_{i_0}$ is a fair execution of Algorithm 1 on $\mathcal{P} \setminus \{p\}$. By R3, the only outgoing edge of s in C_{i_0} is $\overrightarrow{sp}$, so $p'' = s$ is the unique sink of C'_{i_0}. Since s has an outgoing edge $\overrightarrow{sp}$ in C_{i_0}, s is not a sink of C_{i_0} and p is the unique sink in C_{i_0}. ◁

Observe that if s is incident to $\overrightarrow{qs}$ (resp., $\overrightarrow{rs}$), then since qp (resp., rp) is stable, by R4, p is incident to $\overrightarrow{qp}$ (resp., $\overrightarrow{rp}$). When among qs and rs, there is one outgoing and one incoming edge, we consider two cases depending on whether p is a sink in $\mathcal{S}_{i_0}$.

▷ Case 2.2. Particle s is incident to $\overrightarrow{qs}$ and $\overrightarrow{sr}$ and p is incident to $\overrightarrow{rp}$.

Proof. Since $\overrightarrow{qs}$ and $\overrightarrow{sp}$ are stable, necessarily $\overrightarrow{qp}$ is stable and p is a sink in $C_{i \geq i_0}$. So, p is never activable in $\mathcal{S}_{i_0}$ and from Lemma 11, q, r, s are never activable in $\mathcal{S}_{i_0}$ either. The edges of $C'_{i \geq i_0} \setminus \{p\}$ have the same orientation as in $C_{i \geq i_0}$ and thus any particle $p' \notin \{p, q, r, s\}$ is activable in C'_i if and only if it is activable in C_i. Let us consider q, r, s. Since $\overrightarrow{sp}$ is between an incoming and an outgoing edge, by Observation 8, R3 is satisfied for s in C'_i. The remaining rules are satisfied for s in C'_i from Observation 9. The arguments for q are the same as in Case 2.1. Due to the incoming edge from the common empty neighbour of r and p and from Observation 8, R3 is satisfied for r in C'_i. The remaining rules are satisfied for r in C'_i from Observation 9. So, q, r, s are never activable in $C'_{i \geq i_0}$. Hence, all rules are satisfied in C'_i and $\mathcal{S}'_{i_0}$ is a fair execution of Algorithm 1 on $\mathcal{P} \setminus \{p\}$. By R3, the only outgoing edge of r in C_i is $\overrightarrow{rp}$, so $p'' = r$ is the unique sink of C'_i. Since p does not have outgoing edges in $C_{i_1} = C_{i_0}$, r is not a sink of C_{i_0} due to $\overrightarrow{rp}$ and p is the unique sink in C_{i_0}. ◁

▷ Case 2.3. Particle s is incident to $\overrightarrow{qs}$ and $\overrightarrow{sr}$ and p is incident to $\overrightarrow{pr}$.

Proof. As noted before the stable edge qp is oriented as $\overrightarrow{qp}$ by R4. By Lemma 11, p, q and s are never activable in $\mathcal{S}_{i_0}$ and the edges pr and sr are stable. The edges of $C'_{i\geq i_0} \setminus \{p\}$ have the same orientation as in $C_{i\geq i_0}$. So, for any $p' \notin \{p,q,s\}$, p' is activable in C'_i if and only if it is activable in C_i. Let us consider q and s. Using the same arguments for both q and s as in Case 2.2, we obtain that R1–R4 are always satisfied at q and s, and that they are never activable in $C'_{i\geq i_0}$. Hence, $\mathcal{S}'_{i_0}$ is a fair execution of Algorithm 1 on $\mathcal{P} \setminus \{p\}$. The unique sink of C'_{i_0}, $p'' \notin \{q,s\}$, since q and s have outgoing edges in $C \setminus \{p\}$. Since the edges incident to p are stable, C_{i_0} is a final directed configuration. Since p is incident to $\overrightarrow{pr}$ in C_{i_0}, p is not a sink in C_i, hence p'' is the unique sink in C_{i_0}. ◁

▷ Case 2.4. Particle s is incident to $\overrightarrow{sq}$ and $\overrightarrow{sr}$.

Proof. Since pq, pr, ps are stable and since ps is directed as $\overrightarrow{sp}$ in $C_{i\geq i_0}$, p is incident to at most one outgoing edge. Without loss of generality, we can thus assume that $\overrightarrow{qp}$ is in $C_{i\geq i_0}$. Observe that by R3, $\overrightarrow{qp}$ is the only outgoing edge at q and no neighbour of q is activable in $C_{i\geq i_0}$ by Lemma 11. Note also that s has three outgoing edges $\overrightarrow{sq}, \overrightarrow{sp}, \overrightarrow{sr}$ in C_i and since s is not activable in C_i, by R2, all other edges incident to s are incoming. Again, this implies that all neighbours of s different from r are not activable in $C_{i\geq i_0}$. For each $i \geq i_0$, let C^*_i be the configuration obtained from C_i by replacing $\overrightarrow{sq}$ by $\overrightarrow{qs}$.

For any particle $p' \notin \{s,q\}$, R2 and R3 are satisfied at p' in C^*_i since they are satisfied at p' in C_i by Lemma 10 and the orientation of the edges incident to p' in C^*_i is the same as in C_i. Since q only has one outgoing edge in C_i, R2 and R3 are satisfied at q in C^*_i. Since s has three outgoing edges $\overrightarrow{sq}, \overrightarrow{sp}, \overrightarrow{sr}$ in C_i and since all other edges incident to s are incoming, R2 directly holds at s in C^*_i and R3 holds at s in C^*_i since p, q and r are reached through consecutive ports of s by definition. If R4 is not satisfied at some particle p' in C^*_i, there is a directed triangle made of the edges $\overrightarrow{qs}, \overrightarrow{sp'}, \overrightarrow{p'q}$ in C^*_i. Since the only out-neighbours of s in C^*_i are p and r, necessarily, $p' = p$, but this is impossible since pq is oriented from q to p. So, R2, R3, R4 are always satisfied in $C^*_{i\geq i_0}$. Since all edges incident to q and s are stable in C_i, R1 is also satisfied at s and q in C_i and in C^*_i. Hence, q and s are never activable in C^*_i. For any $p' \notin \{q,s\}$, p' is activable in C^*_i if and only if it is activable in C_i. Therefore, $\mathcal{S}^*_i$ is a fair execution of Algorithm 1 on $\mathcal{P}$. Note that when considering $C^*_{i\geq i_0}$, we are in Case 2.2 or 2.3. So, we know that $C^*_{i_0}$ is a final directed configuration that contains a unique sink p'' different from q and s. Since a particle p' is activable in C_{i_0} if and only if it is activable in $C^*_{i_0}$, C_{i_0} is also a final directed configuration, and p'' is the unique sink of C_{i_0}. ◁

Finally, we consider the case where the edges pq and pr are not stable. We remind the reader that from Lemma 12, pq and pr are either both stable or both unstable.

▷ Case 3. The edge between p and s is $\overrightarrow{sp}$ and the edges pq and pr are not stable.

Proof. We will prove that this case is not possible.

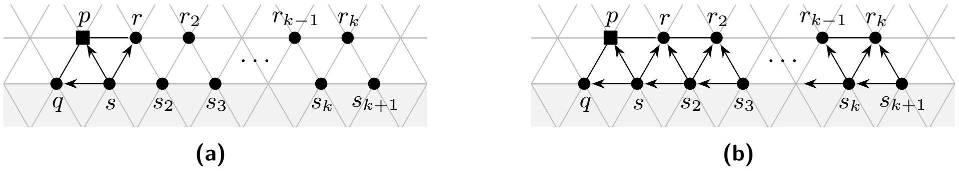


Figure 2 *Left:* The setting in Case 3, that is, a 120° particle p (square) with $\{pq, pr\}$ unstable and s incident to the directed edges sq, sp and sr. *Right:* The final orientation of edges in Case 3.

Without loss of generality, assume that r is on the path connecting p to p^* via 180° particles. Note that it is possible that $p^* = r$. Let $(r_0 = p, r_1 = r, r_2, \dots, r_k = p^*)$ be the path on the boundary from p to p^* whose inner particles are all 180° particles (if $r = p^*$, then $k = 1$). Let s_{j+1} be the common neighbour of any pair of consecutive particles r_j and r_{j+1} with $0 \leq j \leq k-1$, and observe that $s_1 = s$. Let $s_0 = q$ and let s_{k+1} be the neighbour of r_k on the boundary that is distinct from r_{k-1}. This setting is also shown in Figure 2a.

Since s is incident to a stable edge, $\overrightarrow{sp}$, from Lemma 11 all edges incident to s are stable. By Lemma 11 applied at q and r and since pq and pr are not stable, necessarily $\overrightarrow{sq}$ and $\overrightarrow{sr}$ are stable in $\mathcal{S}_{i_0}$. Since in this setting s has three outgoing edges, sq, sp and sr, from R2 ss_2 is incoming to s. So, s_2 is incident to the stable outgoing edge $\overrightarrow{s_2s}$. From Lemma 11, s_2 is not activable and all edges incident to s_2 are stable and directed. From R4, s_2r is oriented from s_2 to r. If $r = p^*$ (i.e., if $k = 1$), r is incident to only one edge that is not stable, which is impossible from Lemma 12. Hence, $k \geq 2$ and rr_2 is not stable. As s_2 is not activable, from Lemma 11, s_2r_2 is stable. If s_2r_2 is directed from r_2 to s_2, r_2 has a stable outgoing edge and from Lemma 11, rr_2 is stable, which is a contradiction for r. So s_2r_2 is directed from s_2 to r_2.

Generalising, for $i \geq 1$ each particle s_i is incident to the stable edges $\overrightarrow{s_is_{i-1}}$, $\overrightarrow{s_ir_{i-1}}$ and $\overrightarrow{s_ir_i}$ and the edge $r_{i-1}r_i$ is not stable. Then, for $i = k$ the edge $r_{k-1}r_k$ should be the only unstable edge incident to r_k which is impossible from Lemma 12, a contradiction. ◁

This ends the proof of Lemma 16. ◀

The proof of Theorem 7 follows from Lemmas 2, 14, 15 and 16. ◀

6 Further Remarks

We showed that our algorithm works assuming that the scheduler is sequential and Gouda fair. The execution presented in Figure 3 shows that if we consider a sequential unfair scheduler (i.e., we only ask that the scheduler activates an activable particle at each step), there exist periodic executions that never reach a valid configuration. It would thus be interesting to understand if we can design a self-stabilising leader election algorithm for simply connected configurations that is correct even with an unfair scheduler. In the case where this is possible, we believe that this would lead to a much more complex algorithm than our algorithm.

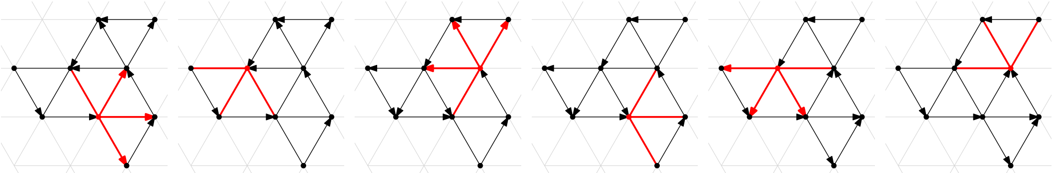


Figure 3 A periodic unfair execution of our algorithm. At each step, the red vertex is activated, and it modifies the status of its incident red edges (i.e., the ones that are not incoming).

Our algorithm heavily uses the geometry of the system and relies on the support being simply connected. For particles that agree on the orientation of the grid, the impossibility results of Dolev et al. [20] no longer hold. One can thus wonder if it is possible to design a silent self-stabilising LE algorithm using constant memory for arbitrary connected configurations if particles agree on the orientation of the grid. Again, the geometry of the grid should be used to overcome the impossibility results of [20], but it seems very challenging.

References

1 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Comput.*, 20(4):279–304, 2007. doi:10.1007/s00446-007-0040-2.

2 Baruch Awerbuch and Rafail Ostrovsky. Memory-efficient and self-stabilizing network reset. In *PODC 1994*, pages 254–263. ACM, 1994. doi:10.1145/197917.198104.

3 Rida A. Bazzi and Joseph L. Briones. Stationary and deterministic leader election in self-organizing particle systems. In *SSS 2019*, volume 11914 of *Lecture Notes in Comput. Sci.*, pages 22–37. Springer, 2019. doi:10.1007/978-3-030-34992-9_3.

4 Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder. Optimal space lower bound for deterministic self-stabilizing leader election algorithms. *Discret. Math. Theor. Comput. Sci.*, 25, 2023. doi:10.46298/dmtcs.9335.

5 Lélia Blin, Pierre Fraigniaud, and Boaz Patt-Shamir. On proof-labeling schemes versus silent self-stabilizing algorithms. In *SSS 2014*, volume 8756 of *Lecture Notes in Comput. Sci.*, pages 18–32. Springer, 2014. doi:10.1007/978-3-319-11764-5_2.

6 Lélia Blin and Sébastien Tixeuil. Compact deterministic self-stabilizing leader election on a ring: the exponential advantage of being talkative. *Distributed Comput.*, 31(2):139–166, 2018. doi:10.1007/s00446-017-0294-2.

7 Paolo Boldi and Sebastiano Vigna. Universal dynamic synchronous self–stabilization. *Distributed Comput.*, 15:137–153, 2002. doi:10.1007/s004460100062.

8 Joseph L. Briones, Tishya Chhabra, Joshua J. Daymude, and Andréa W. Richa. Invited paper: Asynchronous deterministic leader election in three-dimensional programmable matter. In *ICDCN 2023*, pages 38–47. ACM, 2023. doi:10.1145/3571306.3571389.

9 Shukai Cai, Taisuke Izumi, and Koichi Wada. How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. *Theory Comput. Syst.*, 50(3):433–445, 2012. doi:10.1007/s00224-011-9313-z.

10 Jérémie Chalopin, Shantanu Das, and Maria Kokkou. Deterministic leader election for stationary programmable matter with common direction. In *SIROCCO 2024*, volume 14662 of *Lecture Notes in Comput. Sci.*, pages 174–191. Springer, 2024. doi:10.1007/978-3-031-60603-8_10.

11 Jérémie Chalopin, Shantanu Das, and Maria Kokkou. Deterministic self-stabilising leader election for programmable matter with constant memory. *arXiv preprint*, 2024. doi:10.48550/arXiv.2408.08775.

12 Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theor. Comput. Sci.*, 412(40):5541–5561, 2011. doi:10.1016/j.tcs.2010.05.001.

13 Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. The canonical Amoebot model: Algorithms and concurrency control. *Distributed Comput.*, 36(2):159–192, 2023. doi:10.1007/s00446-023-00443-3.

14 Joshua J. Daymude, Andréa W. Richa, and Jamison W. Weber. Bio-inspired energy distribution for programmable matter. In *ICDCN 2021*, pages 86–95. ACM, 2021. doi:10.1145/3427796.3427835.

15 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W Richa, Christian Scheideler, and Thim Strothmann. Amoebot – A new model for programmable matter. In *SPAA 2014*, pages 220–222. ACM, 2014. doi:10.1145/2612669.2612712.

16 Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida A. Bazzi, Andréa W. Richa, and Christian Scheideler. Leader election and shape formation with self-organizing programmable matter. In *DNA 2015*, volume 9211 of *Lecture Notes in Comput. Sci.*, pages 117–132. Springer, 2015. doi:10.1007/978-3-319-21999-8_8.

17 Giuseppe Antonio Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Shape formation by programmable particles. *Distributed Comput.*, 33(1):69–101, 2020. doi:10.1007/s00446-019-00350-6.

18 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974. doi:10.1145/361179.361202.

19 Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000. doi:10.7551/mitpress/6156.001.0001.

20 Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Inf.*, 36(6):447–462, 1999. doi:10.1007/s002360050180.

21 Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. *arXiv preprint*, 2011. doi:10.48550/arXiv.1110.0334.

22 Fabien Dufoulon, Shay Kutten, and William K. Moses Jr. Efficient deterministic leader election for programmable matter. In *PODC 2021*, pages 103–113. ACM, 2021. doi:10.1145/3465084.3467900.

23 Yuval Emek, Shay Kutten, Ron Lavi, and William K Moses Jr. Deterministic leader election in programmable matter. In *ICALP 2019*, volume 132 of *LIPIcs Leibniz Int. Proc. Inform.*, pages 140:1–140:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.140.

24 Sándor P. Fekete, Robert Gmyr, Sabrina Hugo, Phillip Keldenich, Christian Scheffer, and Arne Schmidt. Cadbots: Algorithmic aspects of manipulating programmable matter with finite automata. *Algorithmica*, 83(1):387–412, 2021. doi:10.1007/s00453-020-00761-z.

25 Nicolas Gastineau, Wahabou Abdou, Nader Mbarek, and Olivier Togni. Distributed leader election and computation of local identifiers for programmable matter. In *ALGOSENSORS 2018*, volume 11410 of *Lecture Notes in Comput. Sci.*, pages 159–179. Springer, 2018. doi:10.1007/978-3-030-14094-6_11.

26 Nicolas Gastineau, Wahabou Abdou, Nader Mbarek, and Olivier Togni. Leader election and local identifiers for three-dimensional programmable matter. *Concurr. Comput. Pract. Exp.*, 34(7), 2022. doi:10.1002/cpe.6067.

27 Mohamed G. Gouda. The theory of weak stabilization. In *WSS 2001*, volume 2194 of *Lecture Notes in Comput. Sci.*, pages 114–123. Springer, 2001. doi:10.1007/3-540-45438-1_8.

28 Elliot Hawkes, Byoungkwon An, Nadia M. Benbernou, H. Tanaka, Sangbae Kim, Erik D. Demaine, Daniela Rus, and Robert J. Wood. Programmable matter by folding. *Proc. Natl. Acad. Sci.*, 107(28):12441–12445, 2010. doi:10.1073/pnas.0914069107.

29 Gene Itkis and Leonid Levin. Fast and lean self-stabilizing asynchronous protocols. In *FOCS 1994*, pages 226–239. IEEE Computer Society, 1994. doi:10.1109/SFCS.1994.365691.

30 Gene Itkis, Chengdian Lin, and Janos Simon. Deterministic, constant space, self-stabilizing leader election on uniform rings. In *WDAG 1995*, volume 972 of *Lecture Notes in Comput. Sci.*, pages 288–302. Springer, 1995. doi:10.1007/BFb0022154.

31 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Comput.*, 22(4):215–233, 2010. doi:10.1007/s00446-010-0095-3.

32 Gérard Le Lann. Distributed systems - towards a formal approach. In *IFIP 1977*, pages 155–160. North-Holland, 1977. URL: https://inria.hal.science/hal-03504338.

33 Tommaso Toffoli and Norman Margolus. Programmable matter: Concepts and realization. *Int. J. High Speed Comput.*, 5(2):155–170, 1993. doi:10.1016/0167-2789(91)90296-L.

34 John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966. URL: https://dl.acm.org/doi/book/10.5555/1102024.

35 Damien Woods, Ho-Lin Chen, Scott Goodfriend, Nadine Dabby, Erik Winfree, and Peng Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *ITCS 2013*, pages 353–354. ACM, 2013. doi:10.1145/2422436.2422476.

36 Masafumi Yamashita and Tiko Kameda. Computing on an anonymous network. In *PODC 1988*, pages 117–130. ACM, 1988. doi:10.1145/62546.62568.

Efficient Signature-Free Validated Agreement

Pierre Civit
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Muhammad Ayaz Dzulfikar
NUS Singapore, Singapore

Seth Gilbert
NUS Singapore, Singapore

Rachid Guerraoui
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Jovan Komatovic
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Manuel Vidigueira
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Igor Zablotchi
Mysten Labs, Zürich, Switzerland

Abstract

Byzantine agreement enables n processes to agree on a common L-bit value, despite up to $t > 0$ arbitrary failures. A long line of work has been dedicated to improving the bit complexity of Byzantine agreement in synchrony. This has culminated in COOL, an error-free (deterministically secure against a computationally unbounded adversary) solution that achieves $O(nL + n^2 \log n)$ worst-case bit complexity (which is optimal for $L \geq n \log n$ according to the Dolev-Reischuk lower bound). COOL satisfies strong unanimity: if all correct processes propose the same value, only that value can be decided. Whenever correct processes do not agree *a priori* (there is no unanimity), they may decide a default value $\bot$ from COOL.

Strong unanimity is, however, not sufficient for today's state machine replication (SMR) and blockchain protocols. These systems value progress and require a decided value to always be *valid* (according to a predetermined predicate), excluding default decisions (such as $\bot$) even in cases where there is no unanimity a priori. *Validated Byzantine agreement* satisfies this property (called *external validity*). Yet, the best error-free (or even signature-free) validated agreement solutions achieve only $O(n^2 L)$ bit complexity, a far cry from the $\Omega(nL + n^2)$ Dolev-Reischuk lower bound. Is it possible to bridge this complexity gap?

We answer the question affirmatively. Namely, we present two new synchronous algorithms for validated Byzantine agreement, HashExt and ErrorFreeExt, with different trade-offs. Both algorithms are (1) signature-free, (2) optimally resilient (tolerate up to $t < n/3$ failures), and (3) early-stopping (terminate in $O(f + 1)$ rounds, where $f \leq t$ denotes the actual number of failures). On the one hand, HashExt uses only hashes and achieves $O(nL + n^3\kappa)$ bit complexity, which is optimal for $L \geq n^2\kappa$ (where κ is the size of a hash). On the other hand, ErrorFreeExt is error-free, using no cryptography whatsoever, and achieves $O\big((nL + n^2) \log n\big)$ bit complexity, which is near-optimal for any L.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Validated Byzantine agreement, Bit complexity, Round complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.14

Related Version The full version of this paper, which includes detailed pseudocode and proofs, is available online.
Full Version: `https://arxiv.org/abs/2403.08374v3` [20]

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 14; pp. 14:1–14:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding *Pierre Civit*: Supported in part by the FNS (#40B2-0_218648).
Seth Gilbert: Supported in part by the Singapore MOE Tier 2 grant MOE-T2EP20122-0014.
Manuel Vidigueira: Supported in part by the FNS (#200021_215383).

1 Introduction

Byzantine agreement [42] is arguably the most important problem of distributed computing. It lies at the heart of state machine replication (SMR) [6, 16, 38, 1, 7, 37, 59, 48, 50] and blockchain systems [46, 13, 4, 32, 3, 25, 24]. Additionally, Byzantine agreement plays an essential role in cryptographic protocols such as multi-party computation [33, 11, 36, 10, 30, 17].

Byzantine agreement operates among n processes, out of which up to $t > 0$ can be corrupted by the adversary. A corrupted process is said to be *faulty* and can behave arbitrarily; a non-faulty process is said to be *correct* and follows the prescribed protocol. Let Value denote the set of L-bit values. (As this paper is concerned with multi-valued Byzantine agreement, we set no restrictions on the cardinality of the Value set.) During the agreement protocol, each process *proposes* exactly one value, and eventually the protocol outputs a single *decision*, as per the following interface:

- **request** $\mathsf{propose}(v \in \mathsf{Value})$: a process proposes an L-bit value v.
- **indication** $\mathsf{decide}(v' \in \mathsf{Value})$: a process decides an L-bit value v'.

Intuitively, Byzantine agreement ensures that all correct processes agree on the same *admissible* value. (We formally define the properties of Byzantine agreement in the later part of this section.)

Practical notion of value-admissibility. A critical question in designing practical Byzantine agreement algorithms is which values should be considered admissible. Traditionally, Byzantine agreement algorithms treated the proposals of correct processes as admissible. Consequently, they have focused on properties like *strong unanimity* [5, 18, 52]: if every correct process proposes the same value v, then v is the only possible decision. Notice that in such cases, if even one correct process proposes a value different from the (same) value held by all other $n-1$ processes, it is perfectly legal to decide some default “null op” value (e.g., $\bot$); it is also perfectly legal to decide a value that is “nonsense” from the perspective of the underlying application. Thus, unless all correct processes agree *a priori*, Byzantine agreement algorithms with strong unanimity are not guaranteed to make any “real” progress.

Many modern applications may require a stronger requirement: even if correct processes propose different values, the resulting decision should still adhere to some *validity* test, ensuring that the decision is not “wasted”. Such a condition is usually called *external validity* [14, 41, 45, 56, 5, 61, 31, 44, 55]: any decided value must be valid according to a predetermined logical predicate. We underline that the external validity property is prevalent in today’s blockchain systems. Indeed, as long as a produced block is valid (e.g., no double-spending), the block can safely be added to the chain (irrespectively of who produced it).[1]

[1] Let us underline that real-world blockchain systems might be concerned with *fairness*, thus making the question of “who produced a block” important. However, this work does not focus on fairness (or any similar topic [34, 35]).

Synchronous validated agreement. We study *validated agreement*, a variant of the Byzantine agreement problem satisfying the external validity property, in the standard synchronous setting. Formally, let $\mathsf{valid} : \mathsf{Value} \to \{\mathit{true}, \mathit{false}\}$ be any predetermined predicate. Importantly, correct processes propose valid values. The following properties are guaranteed by validated agreement:

- *Agreement:* No two correct processes decide different values.
- *Integrity:* No correct process decides more than once.
- *Termination:* All correct processes eventually decide.
- *Strong unanimity:* If all correct processes propose the same value v, then no correct process decides any value $v' \neq v$.
- *External validity:* If a correct process decides a value v, then $\mathsf{valid}(v) = \mathit{true}$.

We underline that validated agreement algorithms usually do not satisfy strong unanimity (but only external validity). Additionally, we emphasize that obtaining an agreement algorithm $\mathcal{A}^\star$ that satisfies *both* strong unanimity and external validity is straightforward given (1) an agreement algorithm $\mathcal{A}_1$ satisfying only strong unanimity, and (2) an agreement algorithm $\mathcal{A}_2$ satisfying only external validity. Indeed, to obtain $\mathcal{A}^\star$, processes run $\mathcal{A}_1$ and $\mathcal{A}_2$ in parallel. Then, processes decide (1) the value of $\mathcal{A}_1$ if that value is valid, or (2) the value of $\mathcal{A}_2$ otherwise.

Complexity of synchronous validated agreement. There exist two dominant worst-case complexity metrics when analyzing any synchronous validated agreement algorithm: (1) *the bit complexity*, the total number of bits correct processes send, and (2) *the round complexity*, the number of synchronous rounds it takes for all correct processes to decide (and halt). The lower bound on the bit complexity of validated agreement is $\Omega(nL + n^2)$: (1) the "nL" term comes from the fact that each correct process needs to receive the decided value, and (2) the "n^2" term comes from the seminal Dolev-Reischuk bound [27] stating that even agreeing on a single bit requires $\Omega(n^2)$ exchanged bits. We emphasize that the $\Omega(nL + n^2)$ lower bound holds even in *failure-free* executions in the signature-free world (with signatures, the bound does not hold [56]). The lower bound on the round complexity is $\Omega(f + 1)$ [28], where $f \leq t$ denotes the *actual* number of failures. If an algorithm achieves $O(f + 1)$ round complexity, it is said that the algorithm is *early-stopping.*[2]

State-of-the-art. The most efficient known validated agreement algorithm is Ada-Dare [19]. Ada-Dare achieves $O(nL + n^2\kappa)$ bit complexity (optimal for $L > n\kappa$), where κ denotes a security parameter. However, Ada-Dare internally utilizes threshold signatures [54]. (We emphasize that if $t < n/3$, some partially synchronous authenticated algorithms [60, 15] can trivially be adapted to achieve $O(nL + n^2\kappa)$ bit complexity in synchrony; Ada-Dare tolerates up to $t < n/2$ failures.) Perhaps surprisingly, the best *signature-free* validated agreement algorithms [43, 12, 22, 18] still achieve only $O(n^2L)$ bit complexity, a far cry from the $\Omega(nL + n^2)$ lower bound.

The fact that no efficient signature-free validated agreement is known becomes even more surprising when considering that optimal signature-free algorithms exist for the "traditional" Byzantine agreement problem. COOL [18] is a Byzantine agreement algorithm satisfying (only) strong unanimity while exchanging $O(nL + n^2 \log n)$ bits. Although it was not the goal of the COOL algorithm, COOL can trivially achieve early-stopping (by internally utilizing an early-stopping binary agreement such as [43]). In addition, COOL is optimally resilient

[2] We consider only *asymptotic* early-stopping (as in [43]) instead of *strict* early stopping (as in [28]) that requires termination in exactly $f + 2$ rounds.

(tolerates up to $t < n/3$ failures). Importantly, COOL uses no cryptography whatsoever: we say that COOL is *error-free* as it is deterministically secure against a computationally unbounded adversary.

Is there a fundamental complexity gap between external validity and strong unanimity in the signature-free world? Can signature-free validated agreement be solved efficiently in synchrony? These are the questions we study in this paper.

1.1 Contributions

In this paper, we present the first validated agreement algorithms achieving $o(n^2 L)$ bit complexity *without* signatures:

- First, we introduce HashExt, a hash-based algorithm that exchanges $O(nL + n^3\kappa)$ bits (optimal for $L \geq n^2\kappa$), where κ denotes the size of a hash.
- Second, we provide ErrorFreeExt, an error-free (i.e., cryptography-free) solution that achieves $O\big((nL + n^2)\log n\big)$ bit complexity and is thus nearly-optimal.

Importantly, both HashExt and ErrorFreeExt are (1) optimally resilient (tolerate up to $t < n/3$ failures), and (2) early-stopping (terminate in $O(f + 1)$ synchronous rounds). A comparison of our new algorithms with the state-of-the-art can be found in Table 1.

Table 1 Performance of deterministic synchronous agreement algorithms with L-bit values and κ-bit security parameter. S stands for "strong unanimity", E stands for "external validity", and IC stands for "interactive consistency" (where processes agree on the proposals of all processes). (There exists a trivial reduction from IC to S + E, where each correct process decides the most represented valid value in the decided vector. Hence, we write that IC implies S + E.) All considered algorithms are early-stopping, except for Ada-Dare$_{ic}$ and Ada-Dare$_{su}$ (whose goal was not early-stopping).

Protocol	Validity	Bit complexity	Resilience	Cryptography
COOL [18, 43]	S	$O(nL + n^2 \log n)$	$n > 3t$	None
Parallel COOL [18, 43]	IC → (S + E)	$O(n^2 L + n^3 \log n)$	$n > 3t$	None
Ada-Dare$_{ic}$ [19]	IC → (S + E)	$O(n^2 L + n^2\kappa)$	$n > 2t$	Threshold Sign.
Ada-Dare$_{su}$ [19]	S + E	$O(nL + n^2\kappa)$	$n > 2t$	Threshold Sign.
HashExt	S + E	$O(nL + n^3\kappa)$	$n > 3t$	Hash
ErrorFreeExt	S + E	$O\big((nL + n^2)\log n\big)$	$n > 3t$	None
Lower bound [27, 21]	Any	$\Omega(nL + n^2)$	$t \in \Omega(n)$	Any

1.2 Overview & Technical Challenges

Why is efficient validated agreement hard? To solve the validated agreement problem (i.e., to satisfy external validity), a decided value must be valid. Therefore, a validated agreement algorithm needs to ensure that it is operating on (or converging to) a valid value. If the value (in its entirety) is attached to every message, satisfying external validity is (relatively) simple: each message can be individually validated and invalid messages can be ignored. Unfortunately, attaching an L-bit value to each message is inherently expensive, yielding a sub-optimal bit complexity of $\Omega(n^2 L)$.

To avoid attaching an L-bit value to each message, the most efficient solutions to validated agreement (designed for arbitrary-sized values) involve coding techniques, where an L-bit value is split into n different shares of $O(\frac{L}{n} + \log n)$ size. The goal is to (somehow) reach agreement on a valid value using $O(n^2)$ messages of $O(\frac{L}{n} + \log n)$ bits, for a total of $O(nL + n^2 \log n)$ exchanged bits. However, this "coding-based" design introduces a new challenge. How can a process that only holds one share (or constantly many shares) know that the corresponding value is valid? For example, to check if a split value v is valid, correct processes might attempt to reconstruct it, expending $O(nL + n^2 \log n)$ bits in the process (as reconstruction is expensive). Since there may be (in the worst case) up to $t \in \Omega(n)$ invalid values (from as many faulty processes), this reconstruction process might have to be repeated many times before a valid value is found, resulting in (say) sub-optimal $O(n^2 L + n^3 \log n)$ total communication.

Overview of HashExt. To overview HashExt's design, we first revisit how efficient signature-based validated agreement is solved (see, e.g., [19]). In the signature-based paradigm, efficient validated agreement algorithms adopt the following approach: (1) First, each process disseminates its value (using coding techniques) and obtains a *proof of retriveability* (PoR). A PoR is a cryptographic object containing a digest (of a value) and proving that (i) the pre-image of the digest can be retrieved by all correct processes, and (ii) the pre-image of the digest is valid. (2) Second, processes agree on a single PoR. (3) Third, processes retrieve a value corresponding to the agreed-upon PoR. Importantly, each PoR must be "self-certifying": once a correct process obtains an alleged PoR, the process must be able to determine if the PoR is valid to be sure that if this PoR gets decided in the second step, a valid value can be retrieved. That is why PoRs are usually implemented using signatures: if a PoR contains a *signature-based certificate*, processes can be confident in its validity. Due to this "self-certifying" nature of PoRs, it seems challenging to adapt them to the signature-free world.

To design a hash-based validated agreement algorithm HashExt, we (roughly) follow the aforementioned three-step approach with one fundamental difference: HashExt utilizes *implicit* ("non-self-certyfing") PoRs. Given any observed digest d, a correct process executing HashExt can determine if (1) the pre-image v of digest d can be retrieved, and (2) v is valid. There is no *proof* that the valid pre-image can be retrieved – only the protocol design ensures this guarantee.

Overview of ErrorFreeExt. To implement ErrorFreeExt, our error-free (cryptography-free) near-optimal solution, we rely on a recursive structure – carefully adapting to long values the recursive design proposed by [12, 22, 43, 51] that is only concerned with constant-sized values. At each recursive iteration with n processes, processes are statically partitioned into two halves that run the algorithm among $n/2$ processes. Moreover, each recursive iteration exhibits "additional work" through the *graded consensus* [9, 2] primitive. Intuitively, the graded consensus primitive reconciles decisions made by two distinct halves to ensure that all processes agree on a unique valid value. Due to the recursive nature of ErrorFreeExt, its bit complexity depends on the complexity of graded consensus. To obtain ErrorFreeExt's near-optimal $O\big((nL+n^2) \log n\big)$ bit complexity, we observe that a graded consensus algorithm with $O(nL + n^2 \log n)$ bits can be derived from the "reducing" technique introduced by the previously mentioned COOL [18] protocol.[3]

Roadmap. We define the system model and introduce some preliminaries in §2. We present HashExt in §3, whereas ErrorFreeExt is introduced in §4. Finally, we conclude in §5. Omitted pseudocode, detailed related work and proofs are relegated to the full version of the paper.

[3] A similar observation has recently been made for (balanced) synchronous *gradecast*, a sender-oriented counterpart to graded consensus [8].

2 System Model & Preliminaries

2.1 System Model

Processes. We consider a static set $\Pi = \{p_1, p_2, ..., p_n\}$ of n processes, where each process acts as a deterministic state machine. Our HashExt (resp., ErrorFreeExt) algorithm implements validated agreement against a computationally bounded (resp., unbounded) adversary that can corrupt up to $t < n/3$ processes at any time during an execution. (We underline that no signature-free agreement algorithm can tolerate $n/3$ or more failures [40], disregarding the restricted-resource model [29] that allows for a higher corruption threshold.) A corrupted process is said to be *faulty*; a non-faulty process is said to be *correct*. We denote by $f \leq t$ the actual number of faulty processes; we emphasize that f is not known.

Stopping. Each correct process can invoke a special stop request while executing any protocol. Once a correct process stops executing a protocol, it ceases taking any steps (e.g., sending and receiving messages).

Communication network. Processes communicate by exchanging messages over an authenticated point-to-point network. The communication network is reliable: if a correct process sends a message to a correct process, the message is eventually received.

Synchrony. We assume the standard synchronous environment in which the computation unfolds in synchronous δ-long rounds, where δ denotes the known upper bound on message delays. In each round $1, 2, ... \in \mathbb{N}$, each process (1) performs (deterministic) local computations, (2) sends (possibly different) messages to (a subset of) the other processes, and (3) receives the messages sent to it by the end of the round.

2.2 Complexity Measures

Let Agreement be any synchronous validated agreement algorithm, and let $\mathcal{E}(\text{Agreement})$ denote the set of Agreement's executions. Let $\alpha \in \mathcal{E}(\text{Agreement})$ be any execution. The bit complexity of α is the number of bits correct processes collectively send throughout α. The *bit complexity* of Agreement is then defined as

$$\max_{\alpha \in \mathcal{E}(\text{Agreement})} \bigg\{ \text{the bit complexity of } \alpha \bigg\}.$$

Similarly, the latency complexity of α is the time it takes for all correct processes to decide and stop in α. The *latency complexity* of Agreement is then defined as

$$\max_{\alpha \in \mathcal{E}(\text{Agreement})} \bigg\{ \text{the latency complexity of } \alpha \bigg\}.$$

We say that Agreement satisfies *early stopping* if and only if the latency complexity of Agreement belongs to $O\big((f+1)\delta\big)$. Note that the maximum number of rounds Agreement requires to decide – the *round complexity* of Agreement – is equal to the latency complexity of Agreement divided by δ. Throughout the paper, we use the latency and round complexity interchangeably.

2.3 Building Blocks

This subsection overviews building blocks utilized in both HashExt and ErrorFreeExt.

Reed-Solomon codes. HashExt and ErrorFreeExt rely on Reed-Solomon (RS) codes [53]. We use RSEnc and RSDec to denote RS' encoding and decoding algorithms. In brief, $\mathsf{RSEnc}(M, m, k)$ takes as input a message M consisting of k symbols, treats it as a polynomial of degree $k-1$, and outputs m evaluations of the corresponding polynomial. Similarly, $\mathsf{RSDec}(k, r, T)$ takes as input a set of symbols T (some of the symbols might be incorrect) and outputs a degree $k-1$ polynomial (i.e., k symbols) by correcting up to r errors (incorrect symbols) in T. Note that RSDec can correct up to r errors in T and output the original message given that $|T| \geq k + 2r$ [47]. Importantly, the bit-size of an RS symbol obtained by the $\mathsf{RSEnc}(M, m, k)$ algorithm is $O(\frac{|M|}{k} + \log m)$, where $|M|$ denotes the bit-size of the message M.

Graded consensus. Both HashExt and ErrorFreeExt make extensive use of the graded consensus primitive [9, 2] (also known as Adopt-Commit [26]), whose formal specification is given in Module 1. In brief, graded consensus allows processes to propose their input value from the GC_Value set and decide on some value from the GC_Value set with some binary grade. The graded consensus primitive ensures agreement among correct processes only if some correct process decides a value with (higher) grade 1. If no correct process decides with grade 1, graded consensus allows correct processes to disagree. (Thus, graded consensus is a weaker problem than validated agreement.) HashExt employs the graded consensus primitive on hash values (GC_Value $\equiv$ the set of all hash values). On the other hand, ErrorFreeExt utilizes graded consensus on values proposed to validated agreement (GC_Value $\equiv$ Value).

Module 1 Graded consensus.

Events:

- *request* $\mathsf{propose}(v \in \mathsf{GC\_Value})$: a process proposes a value $v \in \mathsf{GC\_Value}$.
- *indication* $\mathsf{decide}(v' \in \mathsf{GC\_Value}, g' \in \{0, 1\})$: a process decides a value $v' \in \mathsf{GC\_Value}$ with a grade g'.

Assumed behavior:

- Every correct process proposes exactly once.
- All correct processes propose simultaneously (i.e., in the same round). (We revisit this assumption for the graded consensus primitive employed in ErrorFreeExt; see §4.3.)

Properties:

- *Strong unanimity:* If all correct processes propose the same value v and a correct process decides a pair (v', g'), then $v' = v$ and $g' = 1$.
- *Justification:* If a correct process decides a pair $(v', \cdot)$, then v' was proposed by a correct process.
- *Consistency:* If any correct process decides a pair $(v, 1)$, then no correct process decides any pair $(v' \neq v, \cdot)$.
- *Integrity:* No correct process decides more than once.
- *Termination:* All correct processes decide simultaneously (i.e., in the same round). (The "simultaneous" termination is revisited in the graded consensus primitive employed in ErrorFreeExt; see §4.3.)

3 HashExt: Optimal Early-Stopping Hash-Based Solution

In this section, we present HashExt, our hash-based validated Byzantine agreement solution that achieves $O(nL + n^3\kappa)$ bit complexity, which is optimal for $L \geq n^2\kappa$ (κ denotes the size of a hash value). Additionally, HashExt is (1) optimally resilient as it tolerates up to $t < n/3$ faults, and (2) early-stopping as it terminates in $O\big((f+1)\delta\big)$ time (i.e., $O(f+1)$ synchronous rounds).

We start by introducing the building blocks of HashExt (§3.1). Then, we present HashExt's pseudocode (§3.2). Finally, we present a proof sketch of HashExt's correctness and complexity (§3.3). We relegate a proof of HashExt's correctness and complexity to the full version of the paper.

3.1 Building Blocks

Digests. We assume a collision-resistant function $\mathsf{digest} : \mathsf{Value} \to \mathsf{Digest} \equiv \{0,1\}^\kappa$, where κ is a security parameter. Concretely, the $\mathsf{digest}(v \in \mathsf{Value})$ function performs the following steps: (1) it encodes value v into n RS symbols $[m_1, m_2, ..., m_n] \leftarrow \mathsf{RSEnc}(v, n, t+1)$; (2) it aggregates $[m_1, m_2, ..., m_n]$ into an accumulation value z_v using the Merkle-tree-based (i.e., hash-based) cryptographic accumulator [49]; (3) it returns z_v. Note that, as we employ hash-based Merkle trees, an accumulation value z_v is a hash. The formal definition of the $\mathsf{digest}(\cdot)$ function can be found in the full version of the paper.

Data dissemination. The formal specification of the data dissemination primitive is given in Module 2. Intuitively, the data dissemination primitive allows all correct processes to obtain the same value $v^\star$ assuming that (1) all correct processes a priori agree on the digest $d^\star$ of value $v^\star$ (even if processes do not know the pre-image $v^\star$ of $d^\star$ a priori), and (2) at least one correct process initially holds the pre-image $v^\star$. We relegate the implementation of the data dissemination primitive to the full version of the paper. In brief, the implementation heavily relies on Merkle-tree-based accumulators and it exchanges $O(nL + n^2\kappa \log n)$ bits while terminating in 2δ time.

Module 2 Data dissemination.

Events:

- *request* $\mathsf{input}(v \in \mathsf{Value} \cup \{\bot\}, d \in \mathsf{Digest})$: a process inputs a value v (or $\bot$) and a digest d.
- *request* $\mathsf{output}(v' \in \mathsf{Value})$: a process outputs a value v'.

Assumed behavior:

- All correct processes input a pair. We underline that correct processes might not input their values simultaneously (i.e., at the exact same round).
- No correct process stops unless it has previously output a value.
- There exists a value $v^\star \in \mathsf{Value}$ ($v^\star \neq \bot$) and a digest $d^\star = \mathsf{digest}(v^\star)$ such that:
  - If any correct process inputs a pair $(v \in \mathsf{Value}, \cdot)$, then $v = v^\star$.
  - If any correct process inputs a pair $(\cdot, d \in \mathsf{Digest})$, then $d = d^\star$.
  - At least one correct process inputs a pair $(v^\star, d^\star)$.

Properties:

- *Safety:* If any correct process outputs a value v, then $v = v^\star$.
- *Liveness:* Let τ be the first time by which all correct processes have input a pair. Then, every correct process outputs a value by time $\tau + 2\delta$.
- *Integrity:* No correct process outputs a value unless it has previously input a pair.

3.2 Pseudocode

The pseudocode of HashExt is given in Algorithm 1.

Key idea. The crucial idea behind HashExt is to ensure that all correct processes agree on a digest $d^\star$ of a *valid* value $v^\star$ such that at least *one* correct process knows the pre-image $v^\star$ of $d^\star$. To solve validated agreement, it then suffices to utilize the data dissemination primitive (see Module 2): if (1) all correct processes input the same digest $d^\star$, and (2) at least one correct process inputs the pre-image $v^\star$ of $d^\star$, then all correct processes agree on the (valid) value $v^\star$. Given that the data dissemination primitive exchanges $O(nL + n^2\kappa \log n)$ bits and terminates in 2 rounds, HashExt dedicates $O(nL + n^3\kappa)$ bits and $O(f+1)$ rounds to agreeing on digest $d^\star$.

Protocol description. HashExt internally utilizes an instance $\mathcal{DD}$ of the data dissemination primitive. We design HashExt in a *view-based* manner: HashExt operates in (at most) $f+1$ views, where each view V has its leader $\mathsf{leader}(V) = p_V$.[4] Each view V internally uses two instances $\mathcal{GC}_1[V]$ and $\mathcal{GC}_2[V]$ of the graded consensus primitive (see Module 1) that operates on digests.

We say that a correct process p_i *commits* a digest d in view V if and only if p_i invokes $\mathcal{DD}.\mathsf{input}(\cdot, d)$ in view V (line 44). Each correct process p_i maintains four important local variables:

- $locked_i$ (line 6): holds a digest (or $\bot$) on which p_i is currently "locked on".
- $vote_i$ (line 7): holds a digest (or $\bot$) currently supported by p_i.
- $known\_values_i[D]$, for every digest D (line 9): holds the pre-image of digest D observed by p_i.
- $accepted_i[V]$, for every view V (line 10): holds the set of digest that are "accepted" in view V.

Let p_i be any correct process. Each view V operates in four steps:

1. Process p_i proposes $locked_i$ to $\mathcal{GC}_1[V]$ and decides a pair (d_1, g_1) (line 16). Intuitively, if $d_1 \neq \bot$ and $g_1 = 1$, p_i sticks with digest d_1 throughout the view as it is possible that some other correct process has previously committed digest d_1. (Hence, not sticking with digest d_1 in view V might be dangerous as it could lead to a disagreement on committed digests.)
2. Here, the leader of view V (if correct) aims to enable all correct processes to commit a digest in view V. Specifically, the leader behaves in the following manner:
   - If it decided a non-$\bot$ digest from $\mathcal{GC}_1[V]$, then the leader disseminates the digest (line 20).
   - Otherwise, the leader disseminates its proposal (line 22).

   Process p_i behaves according to the following logic:
   - If p_i decided a non-$\bot$ digest d with grade 1 from $\mathcal{GC}_1[V]$ ($d_1 = d \neq \bot$ and $g_1 = 1$; see the rule at line 23), then p_i supports digest d by broadcasting a SUPPORT message for d (line 24).
   - If p_i decided $\bot$ from $\mathcal{GC}_1[V]$, then p_i supports a digest d by broadcasting a SUPPORT message for d (line 27 or line 30) if (1) it receives digest d from the leader and p_i accepted d in any previous view (line 26), or (2) it receives a valid value v from the leader such that $\mathsf{digest}(v) = d$ (line 28). If the latter case applies, process p_i "observes" the pre-image v of digest d (line 29).
3. Process p_i accepts a digest d in view V if it receives a SUPPORT message for d from $t+1$ processes (line 33). Moreover, process p_i updates its $vote_i$ variable to a digest d if it receives a SUPPORT message for d from $2t+1$ processes (line 35). Otherwise, process p_i sets its $vote_i$ variable to $\bot$ (line 37). Observe that if any correct process p_j updates its $vote_j$ variable to a digest d, then every correct process p_k accepts d in view V. Indeed, as p_j receives a SUPPORT message for digest d from at least $2t+1$ processes out of which at least $t+1$ are correct, it is guaranteed that p_k receives a SUPPORT message for d from at least $t+1$ processes.

[4] HashExt elects leaders in a round-robin fashion.

Algorithm 1 HashExt: Pseudocode (for process p_i).

```
Uses:
    Graded consensus, instances $\mathcal{GC}_1[V]$, $\mathcal{GC}_2[V]$, for each view $V \in [1, t+1]$    ▷ bits: $O(n^2\kappa)$; rounds: 2
    Data dissemination, instance $\mathcal{DD}$    ▷ bits: $O(nL + n^2\kappa \log n)$; rounds: 2
Local variables:
    Value $v_i \leftarrow p_i$'s proposal
    Digest $\mathit{locked}_i \leftarrow \bot$    ▷ locked digest
    Digest $\mathit{vote}_i \leftarrow \bot$    ▷ digest to be voted for
    View $\mathit{committed\_view}_i \leftarrow \bot$
    Map(Digest $\to$ Value) $\mathit{known\_values}_i \leftarrow \{\bot, \bot, ..., \bot\}$    ▷ values corresponding to digests
    Map(View $\to$ Set(Digest)) $\mathit{accepted}_i \leftarrow \{\emptyset, \emptyset, ..., \emptyset\}$    ▷ accepted digests per view
– Task 1 –
for each view $V \in [1, t+1]$:
    if $\mathit{committed\_view}_i \neq \bot$ and $\mathit{commited\_view}_i + 1 = V$: complete the view after 6 synchronous rounds
    if $\mathit{commited\_view}_i \neq \bot$ and $V > \mathit{committed\_view}_i + 1$: do not execute the view
    Step 1 of view $V$:    ▷ 2 synchronous rounds
        Let $(d_1 \in \text{Digest} \cup \{\bot\}, g_1 \in \{0, 1\}) \leftarrow \mathcal{GC}_1[V].\text{propose}(\mathit{locked}_i)$
    Step 2 of view $V$:    ▷ 2 synchronous round
        if $p_i = \text{leader}(V)$:
            if $d_1 \neq \bot$:    ▷ check if a non-$\bot$ digest is decided from $\mathcal{GC}_1[V]$
                broadcast $d_1$    ▷ broadcast a non-$\bot$ digest decided from $\mathcal{GC}_1[V]$
            else:
                broadcast $v_i$    ▷ broadcast the proposed value
        if $d_1 \neq \bot$ and $g_1 = 1$:
            broadcast $\langle \text{SUPPORT}, d_1 \rangle$
        else:
            if $d_l \in$ Digest is received from $\text{leader}(V)$ and a view $V' < V$ exists with $d_l \in \mathit{accepted}[V']$:
                broadcast $\langle \text{SUPPORT}, d_l \rangle$
            else if $v_l \in$ Value is received from $\text{leader}(V)$ such that $\text{valid}(v_l) = \mathit{true}$:
                $\mathit{known\_values}[\text{digest}(v_l)] \leftarrow v_l$
                broadcast $\langle \text{SUPPORT}, \text{digest}(v_l) \rangle$
    Step 3 of view $V$:    ▷ 0 synchronous round (only local computation)
        if exists $d \in$ Digest such that a $\langle \text{SUPPORT}, d \rangle$ message is received from $t + 1$ processes:
            $\mathit{accepted}_i[V] \leftarrow \mathit{accepted}_i[V] \cup \{d\}$
        if exists $d \in$ Digest such that a $\langle \text{SUPPORT}, d \rangle$ message is received from $2t + 1$ processes:
            $\mathit{vote}_i \leftarrow d$
        else:
            $\mathit{vote}_i \leftarrow \bot$
    Step 4 of view $V$:    ▷ 2 synchronous rounds
        Let $(d_2 \in \text{Digest} \cup \{\bot\}, g_2 \in \{0, 1\}) \leftarrow \mathcal{GC}_2[V].\text{propose}(\mathit{vote}_i)$
        if $d_2 \neq \bot$:    ▷ check if a non-$\bot$ digest is decided from $\mathcal{GC}_2[V]$
            $\mathit{locked}_i \leftarrow d_2$    ▷ digest $d_2$ is locked as some correct process might commit it
            if $g_2 = 1$ and $\mathit{committed\_view}_i = \bot$:    ▷ check if digest $d_2$ is decided with grade 1
                $\mathit{committed\_view}_i \leftarrow V$
                invoke $\mathcal{DD}.\text{input}(\mathit{known\_values}[d_2], d_2)$    ▷ commit digest $d_2$
– Task 2 –    ▷ executed in a separate thread
upon $\mathcal{DD}.\text{output}(v' \in \text{Value})$:
    trigger decide$(v')$
    wait for view $\mathit{committed\_view}_i{+}1$ to be completed (if not yet and if $\mathit{committed\_view}_i{+}1 \leq t{+}1$)
    trigger stop    ▷ process $p_i$ stops HashExt
```

4. Process p_i proposes $vote_i$ to $\mathcal{GC}_2[V]$ and decides a pair (d_2, g_2). If $d_2 \neq \bot$, process p_i updates its $locked_i$ variable to d_2 (line 41). Additionally, if $g_2 = 1$, then p_i commits d_2 (line 44). Importantly, if any correct process p_j commits a digest $d \neq \bot$ in view V, *every* correct process p_k updates its $locked_k$ variable to d. Indeed, as p_j commits d, it decides $(d \neq \bot, 1)$ from $\mathcal{GC}_2[V]$. The consistency property of $\mathcal{GC}_2[V]$ ensures that each correct process p_k decides d from $\mathcal{GC}_2[V]$.

We emphasize that if process p_i commits a digest in some view V, process p_i does not execute any view greater than $V+1$ (line 14). Moreover, if p_i commits in view $V < t+1$, then process p_i necessarily completes view $V+1$ before stopping (line 48). Importantly, process p_i completes view $V+1$ after *exactly* 6 rounds have elapsed. Let us elaborate. As some correct process $p_j \neq p_i$ might never enter view $V+1$ (since it has committed in a view smaller than view V), it is possible that *not all* correct processes participate in view $V+1$. This implies that utilized graded consensus instances might never complete, which further means that process p_i can forever be stuck executing a graded consensus instance in view $V+1$. To avoid this scenario, process p_i completes view $V+1$ after 6 rounds irrespectively of which step of view $V+1$ p_i is in after 6 rounds. Finally, once p_i outputs a value v' from $\mathcal{DD}$ (and completes the aforementioned "next view"), p_i decides v' (line 47) and stops executing HashExt (line 49).

3.3 Proof Sketch

This subsection provides a proof sketch of the following theorem:

▶ **Theorem 1.** *HashExt (Algorithm 1) is a hash-based early-stopping validated agreement algorithm with $O(nL + n^3\kappa)$ bit complexity.*

Our proof sketch focuses on the crucial intermediate guarantees ensured by HashExt.

Preventing disagreement on committed digests. First, we show that correct processes do not disagree on committed digests. Let $\mathcal{V}$ denote the first view in which a correct process commits; let $d^\star$ be the committed digest. No correct process commits any non-$d^\star$ digest in view $\mathcal{V}$ due to the consistency property of $\mathcal{GC}_2[\mathcal{V}]$: it is impossible for correct processes to decide different digests from $\mathcal{GC}_2[\mathcal{V}]$ with grade 1.

If $\mathcal{V} < t+1$, HashExt prevents any non-$d^\star$ digest to be committed in any view greater than $\mathcal{V}$. Specifically, HashExt guarantees that all correct processes commit $d^\star$ (and no other digest) by the end of view $\mathcal{V}+1$. The consistency property of $\mathcal{GC}_2[\mathcal{V}]$ ensures that every correct process p_i updates its $locked_i$ variable to $d^\star$ at the end of view $\mathcal{V}$. Therefore, all correct processes propose $d^\star$ to $\mathcal{GC}_1[\mathcal{V}+1]$, which implies that all correct processes decide $(d^\star, 1)$ from $\mathcal{GC}_1[\mathcal{V}+1]$ (due to the strong unanimity property of $\mathcal{GC}_1[\mathcal{V}+1]$). Hence, all correct processes broadcast a SUPPORT message for digest $d^\star$ (line 24), which further implies that all correct processes propose $d^\star$ to $\mathcal{GC}_2[\mathcal{V}+1]$. Finally, the strong unanimity property of $\mathcal{GC}_2[\mathcal{V}+1]$ ensures that all correct processes decide $(d^\star, 1)$ from $\mathcal{GC}_2[\mathcal{V}+1]$ and thus commit $d^\star$ by the end of view $\mathcal{V}+1$.

Ensuring eventual agreement on the committed digest. Second, we prove that an agreement on the committed digest eventually occurs. Concretely, we now show that *all* correct processes commit a digest by the end of the first view whose leader is correct. Let that view be denoted by $\mathcal{V}_l \in [1, f+1]$ and let $p_{\mathcal{V}_l}$ be the leader of $\mathcal{V}_l$. If any correct process commits a digest in any view smaller than $\mathcal{V}_l$, then all correct processes commit the same digest by the end of view $\mathcal{V}_l$ due to the argument from the previous paragraph. Hence, suppose no correct process commits any digest in any view preceding view $\mathcal{V}_l$. We distinguish two scenarios:

- Let $p_{\mathcal{V}_l}$ decide a digest $d \neq \perp$ from $\mathcal{GC}_1[\mathcal{V}_l]$. Crucially, the justification property of $\mathcal{GC}_1[\mathcal{V}_l]$ ensures that $d \neq \perp$ is proposed by some correct process p_j. Hence, the value of the $locked_j$ variable is d at the beginning of view $\mathcal{V}_l$. Let $V' < \mathcal{V}_l$ denote the view in which p_j updates $locked_j$ to d upon deciding $d \neq \perp$ from $\mathcal{GC}_2[V']$. Again, the justification property of $\mathcal{GC}_2[V']$ guarantees that a correct process proposed d to $\mathcal{GC}_2[V']$ upon receiving $2t+1$ SUPPORT messages for d. As at least $t+1$ such messages are received from correct processes, *every* correct process accepts digest d in view V'.

  In this case, process $p_{\mathcal{V}_l}$ broadcasts digest d in Step 2. We show that all correct processes broadcast a SUPPORT message for digest d. Consider any correct process p_i. We study two possible cases:

  - Let p_i decide a non-$\perp$ digest d' with grade 1 from $\mathcal{GC}_1[\mathcal{V}_l]$. In this case, the consistency property of $\mathcal{GC}_1[\mathcal{V}_l]$ ensures that $d = d'$. Thus, process p_i sends a SUPPORT message for digest d (line 24).
  - Let p_i decide $\perp$ or with grade 0 from $\mathcal{GC}_1[\mathcal{V}_l]$. In this case, process p_i sends a SUPPORT message for digest d (line 27) as (1) it receives d from $p_{\mathcal{V}_l}$, and (2) it accepts d in view $V' < \mathcal{V}_l$.
- Let $p_{\mathcal{V}_l}$ decide $\perp$ from $\mathcal{GC}_1[\mathcal{V}_l]$. Note that this implies that no correct process decides a non-$\perp$ digest with grade 1 from $\mathcal{GC}_1[\mathcal{V}_l]$ (due to the consistency property of $\mathcal{GC}_1[\mathcal{V}_l]$). Hence, process $p_{\mathcal{V}_l}$ broadcasts its valid value v, which then implies that all correct processes send a SUPPORT message for digest $d = \mathsf{digest}(v)$ (line 30).

Hence, there exists a digest d for which all correct processes express their support in both cases. Therefore, all correct processes propose d to $\mathcal{GC}_2[\mathcal{V}_l]$. Finally, the strong unanimity property ensures that all correct processes decide $(d, 1)$ from $\mathcal{GC}_2[\mathcal{V}_l]$ and thus commit digest d in view $\mathcal{V}_l$.

Ensuring that some correct process knows the valid pre-image of the committed digest. We show how HASHEXT enables processes to “obtain” implicit PoRs (see §1). Let $d^\star$ denote the (unique) committed digest. For $d^\star$ to be committed, there exists a correct process that sends a SUPPORT message for $d^\star$ in a view in which $d^\star$ is committed (due to the justification property of $\mathcal{GC}_2[V]$, for every view V). Therefore, it suffices to show that the first correct process to ever send a SUPPORT message for $d^\star$ (or any other digest) does so at line 30 upon receiving valid value $v^\star$ with $\mathsf{digest}(v^\star) = d^\star$. Let p_i denote the first process to send a SUPPORT message for digest $d^\star$ and let it do so in some view V. We study if p_i could have sent the message at lines 24 and 27:

- Process p_i could not have sent the SUPPORT message at line 24 as this would imply that p_i is not the first correct process to send the message for $d^\star$. The justification property of $\mathcal{GC}_1[V]$ ensures that some correct process p_j has its $locked_j$ variable set to $d^\star$ at the beginning of view V. For process p_j to update its $locked_j$ variable to $d^\star$ in some view $V' < V$, there must exist a correct process that sends a SUPPORT message for $d^\star$ in view V' (due to the justification property of $\mathcal{GC}_2[V']$). Therefore, p_i cannot be the first correct process to send a SUPPORT message for $d^\star$.
- Process p_i could not have sent the SUPPORT message at line 27 as this would also imply that p_i is not the first correct process to send the message for $d^\star$. Indeed, for the message to be sent at line 27, process p_i accepts $d^\star$ in some view $V' < V$, which implies that at least one correct process sends a SUPPORT message for $d^\star$ in view V'.

Hence, p_i must have sent the message at line 30, which implies that p_i knows the pre-image $v^\star$ of digest $d^\star$ and that $v^\star$ is valid (due to the check at line 28).

Correctness. The previous three intermediate results show that the preconditions of $\mathcal{DD}$ (see Module 2) are satisfied, which implies that $\mathcal{DD}$ behaves according to its specification. Hence, all correct processes decide the same valid value from HashExt due to the properties of $\mathcal{DD}$.

Complexity. Each view with a non-correct leader exchanges $O(n^2\kappa)$ bits. Moreover, each view with a correct leader exchanges $O(nL + n^2\kappa)$ bits. As $\mathcal{DD}$ exchanges $O(nL + n^2\kappa \log n)$ bits and it is ensured that only $O(1)$ views with correct leaders are executed, HashExt exchanges $O(nL + n^2\kappa) + n \cdot O(n^2\kappa) + O(nL + n^2\kappa \log n) = O(nL + n^3\kappa)$ bits.

As all correct processes start $\mathcal{DD}$ at the end of the first view with a correct leader (at the latest), all correct processes input to $\mathcal{DD}$ in $O(f + 1)$ rounds (recall that each view has 6 rounds). Since $\mathcal{DD}$ guarantees agreement in 2 rounds, all correct decide and stop in $O(f + 1)$ rounds.

On the lack of strong unanimity. Note that HashExt as presented in Algorithm 1 does not satisfy strong unanimity. Indeed, even if all correct processes propose the same value v, it is possible that correct processes agree on a value v' proposed by a faulty leader. However, as specified in §1, it is trivial to modify HashExt to obtain an early-stopping algorithm with both strong unanimity and external validity that exchanges $O(nL + n^3\kappa)$ bits. Indeed, this can be done by running in parallel (1) the current (without strong unanimity) implementation of HashExt, and (2) the error-free early-stopping COOL [18, 43] protocol with only strong unanimity.

4 ErrorFreeExt: Near-Optimal Early-Stopping Error-Free Solution

This section presents ErrorFreeExt, an error-free validated Byzantine agreement algorithm that achieves (1) $O\big((nL + n^2)\log n\big)$ bit complexity, and (2) early stopping. Recall that ErrorFreeExt is also optimally resilient (tolerates up to $t < n/3$ Byzantine processes).

We start by introducing ErrorFreeExt's building blocks (§4.1). To introduce ErrorFreeExt's recursive structure, we first show how (a simplified version of) the recursive structure yields a near-optimal validated agreement without early-stopping – SlowExt (§4.2). Then, we overview ErrorFreeExt (§4.3) and give a proof sketch of its correctness and complexity (§4.4). We relegate ErrorFreeExt's full pseudocode and a formal proof to the full version of the paper.

4.1 Building Blocks

We now overview the building blocks of ErrorFreeExt. Given ErrorFreeExt's recursive structure, the specification of each building block explicitly states its participants (to increase the clarity). Moreover, given that building blocks might be executed among an overly corrupted set of participants (due to the recursion), each building block explicitly states what properties are ensured given the level of corruption among its participants.

Committee broadcast. The formal specification of the committee broadcast primitive is given in Module 3. Committee broadcast is concerned with two sets of processes: (1) $\mathsf{Entire} \subseteq \Pi$, and (2) $\mathsf{Committee} \subseteq \mathsf{Entire}$. Moreover, the primitive is associated with a validated Byzantine agreement algorithm $\mathcal{VA}$ to be executed among processes in $\mathsf{Committee}$. Intuitively, the committee broadcast primitive ensures the following: (1) correct processes in $\mathsf{Committee}$ agree on the same value using the $\mathcal{VA}$ algorithm (given that $\mathsf{Committee}$ is not overly corrupted), and (2) correct processes in Committee disseminate the previously agreed-upon value to all processes in Entire. We underline that the totality property of committee broadcast (deliberately written in orange in Module 3) is important only for ErrorFreeExt's early-stopping, i.e., it can be ignored for SlowExt (in §4.2).

Module 3 Committee broadcast $\langle \mathsf{Entire}, \mathsf{Committee}, \mathcal{VA} \rangle$.

Participants:

- Entire $\subseteq \Pi$; let $x = |\mathsf{Entire}|$ and let x' be the greatest integer smaller than $x/3$.
- Committee $\subseteq$ Entire; let $y = |\mathsf{Committee}|$, let y' be the greatest integer smaller than $y/3$ and let f' be the actual number of faulty processes in Committee.

Utilized validated agreement among Committee:

- $\mathcal{VA}$; let $\mathcal{L}_{\mathcal{VA}}(y, f')$ denote the worst-case latency complexity of $\mathcal{VA}$ with up to f' faulty processes and let $\mathcal{B}_{\mathcal{VA}}(y)$ denote the maximum number of bits any correct process sends while executing $\mathcal{VA}$ with up to y' faulty processes. (We underline that $\mathcal{L}_{\mathcal{VA}}(y, f')$ is based on the non-known *actual* number of failures, whereas $\mathcal{B}_{\mathcal{VA}}(y)$ is based on the known *upper bound* on the number of failures.)

Events:

- *request* input($v \in \mathsf{Value}, g \in \{0, 1\}$): a process inputs a pair (v, g).
- *indication* output($v' \in \mathsf{Value}$): a process outputs a value v'.

Assumed behavior:

- Every correct process inputs a pair.
- If a correct process inputs a pair $(v, \cdot)$, then $\mathsf{valid}(v) = \mathit{true}$.
- No correct process stops unless it has previously output a value.
- If any correct process inputs a pair $(v, 1)$, for any value v, then no correct process inputs a pair $(v' \neq v, \cdot)$.

Properties ensured only if up to x' processes in Entire are faulty:

- *Totality:* Let τ denote the first time at which a correct process outputs a value. Then, every correct process outputs a value by time $\tau + 2\delta$.
- *Stability:* If a correct process inputs a pair $(v, 1)$ and outputs a value v', then $v' = v$.
- *External validity:* If a correct process outputs a value v, then $\mathsf{valid}(v) = \mathit{true}$.
- *Optimistic consensus:* If (1) there are up to y' faulty processes in Committee, and (2) all correct processes in Entire start within 2δ time of each other, the following properties are satisfied:
  - *Liveness:* Let τ be the first time by which all correct processes in Committee have input a pair. Then, every correct process outputs a value by time $\tau + 7\delta + \mathcal{L}_{\mathcal{VA}}(y, f')$.
  - *Agreement:* No two correct processes output different values.
  - *Strong unanimity:* If every correct process proposes a pair $(v, \cdot)$, for any value v, then no correct process outputs a value different from v.

Properties ensured even if more than x' processes in Entire are faulty:

- *Complexity:* Each correct process sends $O(L + x \log x) + \mathcal{B}_{\mathcal{VA}}$ bits.

Finisher. The formal specification of the finisher primitive is given in Module 4. Finisher is executed among a set Entire $\subseteq \Pi$ of processes. Each process inputs a pair $(v \in \mathsf{Value}, g \in \{0, 1\})$, where v is a value and g is a binary grade. In brief, finisher ensures that all correct processes output the same value if all correct processes input the same value with grade 1 (the liveness property). Moreover, finisher ensures totality: if any correct process outputs a value, then all correct processes output the same value. We emphasize that the finisher primitive is introduced *only* for achieving early-stopping in ErrorFreeExt, i.e., it plays no role in SlowExt.

4.2 SlowExt: Achieving Near-Optimality Without Early-Stopping

Wisdom of the ancients. As mentioned in §1.2, the problem with the sequential reconstructive approach is that, by allowing each Byzantine process to impose its own value, we can end up with $f = t \in O(n)$ (wasted) reconstructions of invalid values (with $O(n^2)$ messages each), for a total of $O(n^3)$ messages. Making an analogy to a parliamentary system (e.g., of

Module 4 Finisher $\langle \mathsf{Entire} \rangle$.

Participants:

- $\mathsf{Entire} \subseteq \Pi$; let $x = |\mathsf{Entire}|$ and let x' be the greatest integer smaller than $x/3$.

Events:

- *request* $\mathsf{input}(v \in \mathsf{Value}, g \in \{0,1\})$: a process inputs a pair (v, g).
- *indication* $\mathsf{output}(v' \in \mathsf{Value})$: a process outputs a value v'.

Assumed behavior:

- All correct processes input a pair and they do so within 2δ time of each other.
- No correct process stops unless it has previously output a value.
- If any correct process inputs a pair $(v, 1)$, for any value v, then no correct process inputs a pair $(v' \neq v, \cdot)$.

Properties ensured only if up to x' processes in Entire are faulty:

- *Preservation:* If a correct process p_i outputs a value v', then p_i has previously input a pair $(v', \cdot)$.
- *Agreement:* No two correct processes output different values.
- *Justification:* If a correct process outputs a value, then a pair $(\cdot, 1)$ was input by a correct process.
- *Liveness:* Let all correct processes input a pair $(v, 1)$, for any value v. Let τ be the first time by which all correct processes have input. Then, all correct processes output value v by time $\tau + \delta$.
- *Totality:* Let τ be the first time at which a correct process outputs a value. Then, all correct processes output a value by time $\tau + 2\delta$.

Properties ensured even if more than x' processes in Entire are faulty:

- *Complexity:* Each correct process sends $O(x)$ bits.

some island in ancient Greece [39]), this is the equivalent of allowing every single member of parliament to present their proposal to all others. This is somewhat wasteful. In many modern parliamentary systems, since time is limited, proposals are first filtered *internally* within each party before each party presents *one* proposal to the whole assembly. Hence, no matter how many bad proposals a party might have internally, the whole assembly only discusses one per party. The cost of dealing with bad actors (and proposals) is shifted to the parties, which are individually smaller than the whole assembly. This is (essentially) the crucial realization of [12, 22]. By adopting a recursive framework with two "parties" at each level, [12, 22] obtain non-early-stopping solutions with optimal $O(n^2)$ exchanged messages (albeit still $O(n^2L)$ exchanged bits).

SlowExt in a nutshell. To design SlowExt, we adapt the recursive framework of [12, 22] to long values. More precisely, we follow the recent variant of the framework proposed by [51, 43] that utilizes (1) the graded consensus [9, 2] primitive (instead of the "universal exchange" primitive of [12]; see Module 1), and (2) the committee broadcast primitive (see Module 3). At each recursive iteration, processes are statically partitioned into two halves (according to their identifiers) that run the algorithm among $n/2$ processes (inside that half's committee broadcast primitive) in sequential order. The recursion stops once a validated agreement instance with only a single process is reached; at this point, the process decides its proposal. A graphical depiction of SlowExt is given in the gray part of Figure 1.

Crucially, as $t < n/3$, at least one half contains less than one-third of faulty processes. Therefore, there exists a "healthy" (non-corrupted) half that successfully executes the recursive call (i.e., successfully executes the committee broadcast primitive). However, agreement achieved among a healthy half must be preserved, i.e., preventing an unhealthy half from ruining the "healthy decision" is imperative. To this end, the recursive framework utilizes the graded consensus primitive that allows the correct processes to stick with their previously made (if any) decision. For example, suppose that the first half of processes is healthy. Hence, after executing SlowExt among the first half of processes (i.e., in the first committee broadcast primitive), all correct processes obtain the same value (due to the optimistic

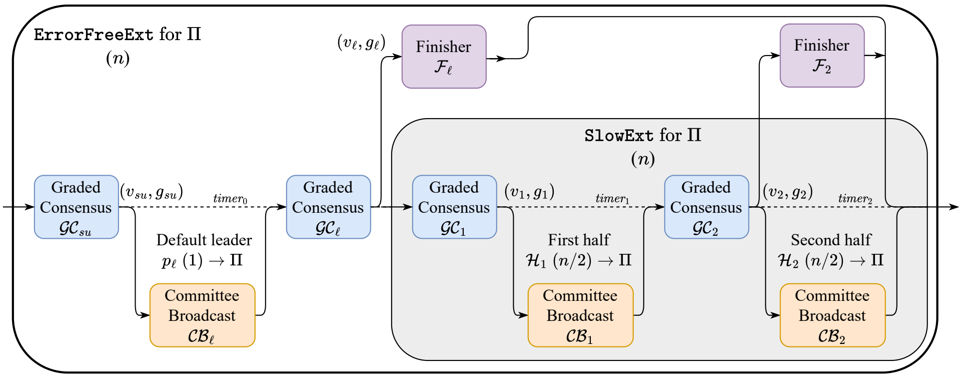


Figure 1 The recursive structure of ErrorFreeExt (and SlowExt).

consensus property of committee broadcast). In this case, the graded consensus primitive $\mathcal{GC}_2$ ensures that correct processes cannot change their values due to the actions of the second half, thus preserving the previously achieved agreement. By implementing both the graded consensus and committee broadcast primitives with only $O(nL + n^2 \log n)$ bits (see the full version of the paper), SlowExt achieves near-optimal asymptotic bit complexity:

$$\sum_{i=0}^{\log n} 2^i \cdot \left(\frac{n}{2^i} L + \left(\frac{n}{2^i}\right)^2 \log\left(\frac{n}{2^i}\right)\right) \leq \sum_{i=0}^{\log n} \left(nL + \frac{n^2}{2^i} \log n\right) \in O\big((nL + n^2) \log n\big).$$

4.3 ErrorFreeExt: Overview

The pseudocode for ErrorFreeExt is provided in the paper's full version and its graphical presentation can be found in Figure 1. Below, we give key insights for obtaining ErrorFreeExt.

Why is SlowExt not early-stopping? SlowExt does not achieve early stopping as SlowExt allocates a predetermined number of rounds for each recursive call: processes cannot *prematurely* terminate a recursive call even if they have already decided. In particular, each recursive call consumes the *maximum* number of rounds necessary for its completion. This maximum number of rounds is proportional to the upper bound t on the number of Byzantine processes rather than the actual number $f \leq t$ of Byzantine processes. As a result, SlowExt incurs round complexity dependent on t rather than f.

From SlowExt to ErrorFreeExt. To achieve early stopping from SlowExt, ErrorFreeExt mirrors the binary approach of [43] and carefully adapts it to long L-bit values. The first key ingredient is the introduction of the finisher instance $\mathcal{F}_2$ that we position (1) before the committee broadcast instance $\mathcal{CB}_2$ led by the second half of processes, and (2) after the graded consensus instance $\mathcal{GC}_2$. In brief, $\mathcal{F}_2$ leverages the presence of the graded consensus instance $\mathcal{GC}_2$ to check if $\mathcal{GC}_2$ ensured agreement among correct processes. If that is the case, then $\mathcal{F}_2$ allows correct processes to terminate immediately (i.e., in $O(\delta)$ time) after the termination of the committee broadcast instance $\mathcal{CB}_1$ led by the first half of processes.

However, the introduction of $\mathcal{F}_2$ to tackle early-stopping brings its share of technical difficulties. Indeed, since the actual number of failures f is unknown, processes cannot remain perfectly synchronized: a correct process p_i might decide (and terminate) at some time τ thinking this is the maximum time before all correct processes decide given the failures p_i observed, whereas another correct process p_j might still be running after time τ as it has observed more failures than p_i. To handle the aforementioned desynchronization, ErrorFreeExt relies on *weak synchronization* ensuring that correct processes execute different sub-modules with at most 2δ desynchronization time: if the first correct process starts executing a sub-module at time τ, then all correct processes start executing the same sub-module by time $\tau + 2\delta$. To achieve this weak synchronization, we follow the standard approach of [57, 58]. Furthermore, to handle the 2δ desynchronization in ErrorFreeExt's sub-modules, we extend the round duration of graded consensus instances from the original δ time to 3δ time. (The specification of the other sub-modules directly tackles the aforementioned desynchronization.) We emphasize that at some point τ, correct processes might be in different rounds: e.g., a correct process p_i can be in round 4, whereas another correct process p_j is in round 5. However, the round duration of 3δ ensures that all correct processes *overlap* in each round for (at least) δ time. As message delays are bounded by δ, the δ-time-overlap is enough to ensure that each correct process hears all r-round-messages from all correct processes before leaving round r. (We emphasize that this is a well-known simulation technique; see, e.g., [43, 23].)

It is important to mention that ErrorFreeExt starts with a single standard "Phase King" iteration: (1) the committee broadcast instance $\mathcal{CB}_l$ with a predetermined leader p_ℓ, (2) the graded consensus instance $\mathcal{GC}_\ell$, and (3) the finisher instance $\mathcal{F}_\ell$. This iteration is added to prevent ErrorFreeExt from running for $\Theta(\log n)$ time when there are only $O(1)$ faults. Indeed, if the predetermined leader p_ℓ is correct, the committee broadcast instance $\mathcal{CB}_\ell$ ensures that all correct processes propose the same valid value v to $\mathcal{GC}_\ell$ in $O(1)$ time after starting ErrorFreeExt. Then, the strong unanimity property of $\mathcal{GC}_\ell$ ensures that all correct processes decide $(v, 1)$ from $\mathcal{GC}_\ell$ and input $(v, 1)$ to $\mathcal{F}_\ell$. This enables $\mathcal{F}_\ell$ to make all correct processes decide v immediately (i.e., in $O(\delta)$ time) after starting.

Finally, the graded consensus instance $\mathcal{GC}_{su}$ (together with $\mathcal{GC}_\ell$) ensures the strong unanimity property. If all correct processes propose the same value v to ErrorFreeExt, then (1) all correct processes decide $(v, 1)$ from $\mathcal{GC}_{su}$ and propose v to $\mathcal{GC}_\ell$, (2) all correct processes decide $(v, 1)$ from $\mathcal{GC}_\ell$ and input $(v, 1)$ to $\mathcal{F}_\ell$, and (3) output v from $\mathcal{F}_\ell$ and decide v from ErrorFreeExt.

4.4 Proof Sketch

This subsection provides a proof sketch of the following theorem:

▶ **Theorem 2.** *ErrorFreeExt is an error-free early-stopping validated agreement algorithm with $O\big((nL + n^2)\log n\big)$ bit complexity.*

We underline that ErrorFreeExt achieves *balanced* bit complexity as its *per-process* complexity is $O\big((L+n)\log n\big)$. This subsection discusses the key intermediate results ensured by ErrorFreeExt.

Gluing all sub-modules together. Processes execute each sub-module within 2δ time of each other, thus enabling the associated implementations to realize the corresponding specifications. The consistency property of the graded consensus primitive ensures a similar consistency for the inputs to the following committee broadcast primitive. Under this condition, the strong unanimity property of the underlying validated agreement protocol ensures agreement if the recursive call is executed with a healthy (non-corrupted) committee.

Ensuring strong unanimity. Strong unanimity is implied by (1) the strong unanimity properties of $\mathcal{GC}_{su}$ and $\mathcal{GC}_\ell$, (2) the stability property of $\mathcal{CB}_\ell$, and (3) liveness and agreement of $\mathcal{F}_\ell$.

Finisher's "lock". If a process decides a value v via a finisher $\mathcal{F} \in \{\mathcal{F}_\ell, \mathcal{F}_2\}$, the justification property of $\mathcal{F}$, combined with the consistency property of the graded consensus $\mathcal{GC} \in \{\mathcal{GC}_\ell, \mathcal{GC}_2\}$ positioned immediately before, ensures that every correct process outputs $(v, \cdot)$ from $\mathcal{GC}$.

From a correct leader or the first healthy committee to a common valid decision. If the predetermined leader p_ℓ is correct, all correct processes agree on a common value after $\mathcal{F}_\ell$: this holds due to (1) the optimistic consensus property of $\mathcal{CB}_\ell$, (2) the strong unanimity property of $\mathcal{GC}_\ell$, and (3) the liveness and agreement properties of $\mathcal{F}_\ell$. Similarly, if p_ℓ is faulty, but the first half of processes is healthy, all correct processes agree on a common value after $\mathcal{F}_2$. Importantly, if some correct process decides via $\mathcal{F}_\ell$, the finisher's lock (see the paragraph above), combined with strong unanimity of $\mathcal{GC}_1$ and $\mathcal{GC}_2$ and the stability property of $\mathcal{CB}_1$, guarantees agreement.

From the second healthy committee to a common valid decision. If a correct process does not decide via $\mathcal{F}_\ell$ or $\mathcal{F}_2$, it means that both the predetermined leader p_ℓ and the first half of processes are unhealthy, which implies that the second half is healthy. If some correct process decides via $\mathcal{F}_2$, the finisher's lock, combined with $\mathcal{CB}_2$'s strong unanimity, preserves agreement. Let us emphasize that if some correct process decides via $\mathcal{F}_\ell$, the agreement is ensured due to (1) the finisher's lock, (2) the strong unanimity properties of $\mathcal{GC}_1$ and $\mathcal{GC}_2$, and (3) the stability property of $\mathcal{CB}_1$.

Complexity. The per-process bit complexity $\mathcal{B}(n)$ of ErrorFreeExt follows from the equation $\mathcal{B}(n) \leq O(L + n\log n) + \max\big(\mathcal{B}(\lfloor \frac{n}{2} \rfloor), \mathcal{B}(\lceil \frac{n}{2} \rceil)\big)$. Similarly, the early stopping property holds due to the following equations: (1) $\mathcal{L}(n, f) \in O(\delta)$ if the predetermined leader p_ℓ is correct, and (2) $\mathcal{L}(n, f) \leq O(\delta) + \mathcal{L}(|\mathcal{H}_1|, f_1) + \mathcal{L}(|\mathcal{H}_2|, f_2)$ otherwise, where f_1 (resp., f_2) denotes the actual number of faulty processes among the first (resp., second) half of processes $\mathcal{H}_1$ (resp., $\mathcal{H}_2$).

5 Concluding Remarks

This paper introduces HashExt and ErrorFreeExt, two synchronous signature-free algorithms for validated Byzantine agreement. Both algorithms are (1) optimally resilient, and (2) early stopping. On one side, HashExt utilizes only collision-resistant hashes, achieving a bit complexity of $O(nL + n^3\kappa)$, which is optimal when $L \geq n^2\kappa$ (with κ being the size of a hash value). Conversely, ErrorFreeExt is error-free, avoids cryptography entirely, and achieves a bit complexity of $O\big((nL + n^2)\log n\big)$, which is nearly optimal for any L. In the future, we plan to focus on the following open questions:

- Is it possible to design an error-free validated agreement algorithm with a bit complexity of $O(nL)$? Our ErrorFreeExt algorithm achieves only $O(nL\log n)$ bit complexity.
- Can HashExt be optimized to achieve $O(nL)$ bit complexity for a wider range of proposal sizes L? Currently, HashExt allows for optimal $O(nL)$ bit complexity only when $L \geq n^2\kappa$.

References

1 Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-Scalable Byzantine Fault-Tolerant Services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005. doi:10.1145/1095810.1095817.

2 Ittai Abraham and Gilad Asharov. Gradecast in synchrony and reliable broadcast in asynchrony with optimal resilience, efficiency, and unconditional security. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 392–398. ACM, 2022. doi:10.1145/3519270.3538451.

3 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solidus: An Incentive-compatible Cryptocurrency Based on Permissionless Byzantine Consensus. *CoRR, abs/1612.02916*, 2016.

4 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus. In James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão, editors, *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, volume 95 of *LIPIcs*, pages 25:1–25:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.OPODIS.2017.25.

5 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 337–346. ACM, 2019. doi:10.1145/3293611.3331612.

6 Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger Wattenhofer. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. In David E. Culler and Peter Druschel, editors, *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. USENIX Association, 2002. URL: http://www.usenix.org/events/osdi02/tech/adya.html.

7 Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *IEEE Trans. Dependable Secur. Comput.*, 7(1):80–93, 2010. doi:10.1109/TDSC.2008.53.

8 Gilad Asharov and Anirudh Chandramouli. Perfect (parallel) broadcast in constant expected rounds via statistical VSS. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part V*, volume 14655 of *Lecture Notes in Computer Science*, pages 310–339. Springer, 2024. doi:10.1007/978-3-031-58740-5_11.

9 Hagit Attiya and Jennifer L. Welch. Multi-valued connected consensus: A new perspective on crusader agreement and adopt-commit. In Alysson Bessani, Xavier Défago, Junya Nakamura, Koichi Wada, and Yukiko Yamauchi, editors, *27th International Conference on Principles of Distributed Systems, OPODIS 2023, December 6-8, 2023, Tokyo, Japan*, volume 286 of *LIPIcs*, pages 6:1–6:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.OPODIS.2023.6.

10 Zuzana Beerliova-Trubiniova and Martin Hirt. Simple and efficient perfectly-secure asynchronous MPC. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4833 LNCS:376–392, 2007. doi:10.1007/978-3-540-76900-2_23.

11 Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988. doi:10.1145/62212.62213.

12 Piotr Berman, Juan A Garay, and Kenneth J Perry. Bit Optimal Distributed Consensus. In *Computer science: research and applications*, pages 313–321. Springer, 1992.

13 Ethan Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. PhD thesis, University of Guelph, 2016. URL: https://atrium.lib.uoguelph.ca/server/api/core/bitstreams/0816af2c-5fd4-4d99-86d6-ced4eef2fb52/content.

14 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and Efficient Asynchronous Broadcast Protocols. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001. doi:10.1007/3-540-44647-8_31.

15 Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet computer consensus. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 81–91. ACM, 2022. doi:10.1145/3519270.3538430.

16 Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4), 2002. doi:10.1145/571637.571640.

17 Nishanth Chandran, Wutichai Chongchitmate, Juan A. Garay, Shafi Goldwasser, Rafail Ostrovsky, and Vassilis Zikas. The hidden graph model: Communication locality and optimal resiliency with adaptive faults. In *ITCS 2015 - Proceedings of the 6th Innovations in Theoretical Computer Science*, pages 153–162, 2015. doi:10.1145/2688073.2688102.

18 Jinyuan Chen. Optimal error-free multi-valued byzantine agreement. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 17:1–17:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.DISC.2021.17.

19 Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. DARE to agree: Byzantine agreement with optimal resilience and adaptive communication. In Ran Gelles, Dennis Olivetti, and Petr Kuznetsov, editors, *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing, PODC 2024, Nantes, France, June 17-21, 2024*, pages 145–156. ACM, 2024. doi:10.1145/3662158.3662792.

20 Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, Manuel Vidigueira, and Igor Zablotchi. Error-free near-optimal validated agreement. *CoRR*, abs/2403.08374, 2024. doi:10.48550/arXiv.2403.08374.

21 Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, Anton Paramonov, and Manuel Vidigueira. All byzantine agreement problems are expensive. In Ran Gelles, Dennis Olivetti, and Petr Kuznetsov, editors, *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing, PODC 2024, Nantes, France, June 17-21, 2024*, pages 157–169. ACM, 2024. doi:10.1145/3662158.3662780.

22 Brian A. Coan and Jennifer L. Welch. Modular Construction of a Byzantine Agreement Protocol with Optimal Message Bit Complexity. *Inf. Comput.*, 97(1):61–85, 1992. doi:10.1016/0890-5401(92)90004-Y.

23 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Make every word count: Adaptive byzantine agreement with fewer words. In Eshcar Hillel, Roberto Palmieri, and Etienne Rivière, editors, *26th International Conference on Principles of Distributed Systems, OPODIS 2022, December 13-15, 2022, Brussels, Belgium*, volume 253 of *LIPIcs*, pages 18:1–18:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.OPODIS.2022.18.

24 Miguel Correia. From Byzantine Consensus to Blockchain Consensus. In *Essentials of Blockchain Technology*, pages 41–80. Chapman and Hall/CRC, 2019.

25 Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: efficient leaderless byzantine consensus and its application to blockchains. In *17th IEEE International Symposium on Network Computing and Applications, NCA 2018, Cambridge, MA, USA, November 1-3, 2018*, pages 1–8. IEEE, 2018. doi:10.1109/NCA.2018.8548057.

26 Carole Delporte-Gallet, Hugues Fauconnier, and Michel Raynal. On the weakest information on failures to solve mutual exclusion and consensus in asynchronous crash-prone read/write systems. *J. Parallel Distributed Comput.*, 153:110–118, 2021. doi:10.1016/J.JPDC.2021.03.015.

27 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, 1985. doi:10.1145/2455.214112.

28 Danny Dolev, Rüdiger Reischuk, and H. Raymond Strong. Early stopping in byzantine agreement. *J. ACM*, 37(4):720–741, 1990. doi:10.1145/96559.96565.

29 Juan Garay, Aggelos Kiayias, Rafail M. Ostrovsky, Giorgos Panagiotakos, and Vassilis Zikas. Resource-Restricted Cryptography: Revisiting MPC Bounds in the Proof-of-Work Era. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12106 LNCS:129–158, 2020. doi:10.1007/978-3-030-45724-2_5.

30 Sanjam Garg, Aarushi Goel, and Abhishek Jain. The broadcast message complexity of secure multiparty computation. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part I*, volume 11921 of *Lecture Notes in Computer Science*, pages 426–455. Springer, 2019. doi:10.1007/978-3-030-34578-5_16.

31 Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In Ittay Eyal and Juan A. Garay, editors, *Financial Cryptography and Data Security - 26th International Conference, FC 2022, Grenada, May 2-6, 2022, Revised Selected Papers*, volume 13411 of *Lecture Notes in Computer Science*, pages 296–315. Springer, 2022. doi:10.1007/978-3-031-18283-9_14.

32 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132757.

33 Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987. doi:10.1145/28395.28420.

34 Vincent Gramoli, Zhenliang Lu, Qiang Tang, and Pouriya Zarbafian. Optimal asynchronous byzantine consensus with fair separability. *IACR Cryptol. ePrint Arch.*, page 545, 2024. URL: https://eprint.iacr.org/2024/545.

35 Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 451–480. Springer, 2020. doi:10.1007/978-3-030-56877-1_16.

36 Hannah Keller, Claudio Orlandi, Anat Paskin-Cherniavsky, and Divya Ravi. MPC with Low Bottleneck-Complexity: Information-Theoretic Security and More. In *4th Conference on Information-Theoretic Cryptography (ITC)*, volume 267, pages 1–21, Aarhus, Denmark, 2023. doi:10.4230/LIPIcs.ITC.2023.11.

37 Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: speculative byzantine fault tolerance. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 45–58. ACM, 2007. doi:10.1145/1294261.1294267.

38 Ramakrishna Kotla and Michael Dahlin. High throughput byzantine fault tolerance. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June*

- *1 July 2004, Florence, Italy, Proceedings*, pages 575–584. IEEE Computer Society, 2004. doi:10.1109/DSN.2004.1311928.

39 Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

40 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982. doi:10.1145/357172.357176.

41 Leslie Lamport, Robert Shostak, and Marshall Pease. Concurrency: The works of leslie lamport. *Association for Computing Machinery*, pages 203–226, 2019.

42 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. doi:10.1145/357172.357176.

43 Christoph Lenzen and Sahar Sheikholeslami. A recursive early-stopping phase king protocol. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 60–69. ACM, 2022. doi:10.1145/3519270.3538425.

44 Yuan Lu, Zhenliang Lu, and Qiang Tang. Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined BFT. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2159–2173. ACM, 2022. doi:10.1145/3548606.3559346.

45 Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-MVBA: Optimal Multi-Valued Validated Asynchronous Byzantine Agreement, Revisited. In Yuval Emek and Christian Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 129–138. ACM, 2020. doi:10.1145/3382734.3405707.

46 Loi Luu, Viswesh Narayanan, Kunal Baweja, Chaodong Zheng, Seth Gilbert, and Prateek Saxena. SCP: A Computationally-Scalable Byzantine Consensus Protocol For Blockchains. *Cryptology ePrint Archive*, 2015.

47 Florence Jessie MacWilliams and Neil James Alexander Sloane. *The Theory of Error-Correcting Codes*, volume 16. Elsevier, 1977.

48 Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1041–1053. ACM, 2019. doi:10.1145/3319535.3354225.

49 Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987. doi:10.1007/3-540-48184-2_32.

50 Atsuki Momose and Ling Ren. Multi-threshold byzantine fault tolerance. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 1686–1699. ACM, 2021. doi:10.1145/3460120.3484554.

51 Atsuki Momose and Ling Ren. Optimal Communication Complexity of Authenticated Byzantine Agreement. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 32:1–32:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.DISC.2021.32.

52 Kartik Nayak, Ling Ren, Elaine Shi, Nitin H. Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*,

volume 179 of *LIPIcs*, pages 28:1–28:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.28.

53 Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

54 Victor Shoup. Practical threshold signatures. In Bart Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 207–220. Springer, 2000. doi:10.1007/3-540-45539-6_15.

55 Anping Song and Cenhao Zhou. Flexbft: A flexible and effective optimistic asynchronous bft protocol. *Applied Sciences*, 14(4):1461, 2024.

56 Alexander Spiegelman. In search for an optimal authenticated byzantine agreement. *arXiv preprint arXiv:2002.06993*, 2020.

57 T. K. Srikanth and Sam Toueg. Optimal clock synchronization. In Michael A. Malcolm and H. Raymond Strong, editors, *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, August 5-7, 1985*, pages 71–86. ACM, 1985. doi:10.1145/323596.323603.

58 T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *J. ACM*, 34(3):626–645, 1987. doi:10.1145/28869.28876.

59 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1):16–30, 2013. doi:10.1109/TC.2011.221.

60 Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. Dispersedledger: High-throughput byzantine consensus on variable bandwidth networks. In Amar Phanishayee and Vyas Sekar, editors, *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 493–512. USENIX Association, 2022. URL: https://www.usenix.org/conference/nsdi22/presentation/yang.

61 You Zhou, Zongyang Zhang, Haibin Zhang, Sisi Duan, Bin Hu, Licheng Wang, and Jianwei Liu. Dory: Asynchronous BFT with reduced communication and improved efficiency. *IACR Cryptol. ePrint Arch.*, page 1709, 2022. URL: https://eprint.iacr.org/2022/1709.

Convex Consensus with Asynchronous Fallback

Andrei Constantinescu
ETH Zürich, Switzerland

Diana Ghinea
ETH Zürich, Switzerland

Roger Wattenhofer
ETH Zürich, Switzerland

Floris Westermann
ETH Zürich, Switzerland

Abstract

Convex Consensus (CC) allows a set of parties to agree on a value v inside the convex hull of their inputs with respect to a predefined abstract convexity notion, even in the presence of byzantine parties. In this work, we focus on achieving CC in the best-of-both-worlds paradigm, i.e., simultaneously tolerating at most t_s corruptions if communication is synchronous, and at most $t_a \leq t_s$ corruptions if it is asynchronous. Our protocol is randomized, which is a requirement under asynchrony, and we prove that it achieves optimal resilience. In the process, we introduce communication primitives tailored to the network-agnostic model. These are a deterministic primitive allowing parties to obtain intersecting views (*Gather*), and a randomized primitive leading to identical views (*Agreement on a Core-Set*). Our primitives provide stronger guarantees than previous counterparts, making them of independent interest.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols

Keywords and phrases convex consensus, network-agnostic protocols, agreement on a core-set

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.15

Related Version *Full Version*: https://ia.cr/2023/1364

Acknowledgements We thank Julian Loss and the anonymous reviewers for their useful suggestions.

1 Introduction

Arranging a meeting place for a group of n people in a city is a common problem, as determining a location that is convenient and accessible for everyone can be challenging. While locations can be determined by their geographic coordinates, we need to prevent agreement on the coordinates of a restricted area, e.g., some private property. Hence, it may be more realistic to represent the city as a graph, with streets modeled as edges and intersections as vertices. Participants are initially in different locations (i.e., vertices), and they want to agree on a vertex for the meeting point using pair-wise communication channels. Finding such a meeting point, while also considering that some of the participants may choose not to follow the protocol, describes the Convex Consensus problem (CC).

The CC problem serves as a unifying framework for various agreement problems that deal with different input spaces. Such input spaces may be continuous, such as $\mathbb{R}^D$, or discrete, such as graphs and even lattices. Essentially, CC assumes a publicly available input space V (this could be the set of locations) equipped with a convexity notion $\mathcal{C}$ (roughly meant to formalize which meeting points are convenient with respect to the participants' inputs). For example, in the case of $\mathbb{R}^D$, the standard "straight-line" convexity notion can be considered. In contrast, convexity notions for graphs may be defined in various ways: for example, *geodesic convexity*, defined over shortest paths between vertices, or *monophonic convexity*,

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 15; pp. 15:1–15:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

defined over minimal/chordless paths. For a given convexity notion, CC is concerned with enabling parties to agree on a value in the convex hull of their inputs. This should be achieved even if up to t of the parties are corrupted (byzantine) and may exhibit malicious behavior.

A natural question to ask is *"For which values of t can CC be achieved?"*. Prior work provides an almost complete answer for *the synchronous model*, i.e., where the parties' clocks are synchronized and messages get delivered within a known amount of time Δ. In this model, the solvability of CC depends on the structure of the input space: concretely, on the space's Helly number ω (e.g., $D+1$ for $\mathbb{R}^D$ with straight-line convexity). CC can be solved in the synchronous model if $t < n/\omega$, and this condition is also necessary for convex geometries (a restricted class of convexity spaces) and for $\mathbb{R}^D$ with straight-line convexity [33, 37].

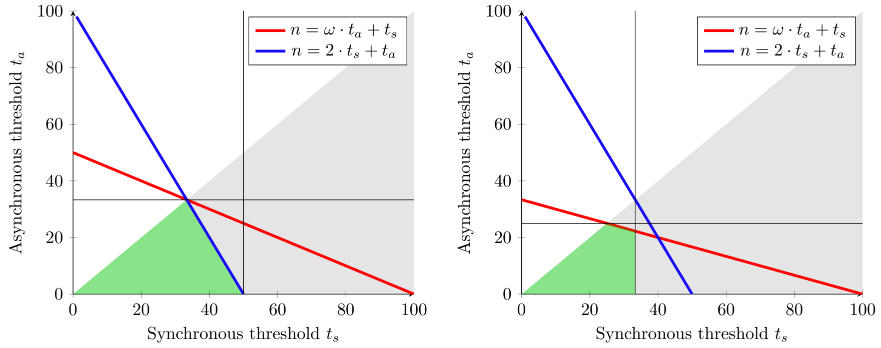
One may argue that the synchronous model's assumptions are too strong: in practice, the maximum delay Δ will often be violated during times of increased network load or outages. A well-established alternative is the *asynchronous model.* This only assumes that parties' messages get delivered eventually, leading to protocols that are highly robust to adverse network conditions. The solvability of CC in this model has only been partly characterized so far: for convex geometries and $\mathbb{R}^D$ with straight-line convexity, the condition $t < n/(\omega+1)$ is necessary [33, 37]. This is, however, only known to be sufficient in the asynchronous model for a relaxed version of CC which allows parties to agree up to some error (Approximate Agreement, AA), and only on particular input spaces. We highlight that the asynchronous model comes with an intrinsic limitation: even if the condition $t < n/(\omega+1)$ were to be proven sufficient for achieving asynchronous CC in all convexity spaces, there is a gap between this threshold and the $t < n/\omega$ threshold sufficing for synchronous networks. That is, an asynchronous CC protocol (which would have to be randomized, due to [18]) would achieve its guarantees regardless of the network conditions, but at the expense of tolerating a lower number of corruptions in comparison to synchronous alternatives.

This is where a third model steps in: *the network-agnostic model*, introduced by Blum, Katz, and Loss [9], which aims to combine the advantages of both established models. This model has gained significant popularity in recent years, and has covered problems such as Byzantine Agreement [9, 15], AA on real and multidimensional values [20, 21], State-Machine Replication [10] and Multi-Party Computation [6, 11, 15]. Concretely, given two thresholds $t_a \leq t_s$, a network-agnostic protocol should tolerate t_s corruptions if the network is synchronous and t_a if it is asynchronous, without knowing which of the two happens to be the case. We add that such a network-agnostic protocol with $t_a = 0$ still provides superior guarantees in comparison to a synchronous counterpart: it maintains its properties even if the synchrony assumptions fail provided that no party is corrupted.

Our work will primarily investigate the necessary and sufficient conditions for achieving CC for arbitrary convexity spaces in the *network-agnostic model.* We provide a complete characterization, showing that the condition $n > \max(\omega \cdot t_s, \omega \cdot t_a + t_s, 2 \cdot t_s + t_a)$ is necessary and sufficient in any convexity space with $\omega > 1$. This condition, illustrated in Figure 1, allows for a trade-off between the two expected optimal resilience bounds of the pure models (which we additionally prove to be tight leading up to our main result). We add that $\omega = 1$ refers to convexity spaces where some $v \in V$ is contained in all non-empty convex sets, allowing for trivial protocols (parties may simply output v).

1.1 Our contributions

Impossibility results. We first prove that the condition $n > \max(\omega \cdot t_s, \omega \cdot t_a + t_s, 2 \cdot t_s + t_a)$ is necessary. We additionally generalize the aforementioned lower bounds from convex geometries and $\mathbb{R}^D$ to all convexity spaces, so that $t < n/\omega$ is required for the synchronous case and $t < n/(\omega+1)$ for the asynchronous one. For these proofs, we define *adversarial families*, which will allow us to derive general scenario-based arguments [32].



(a) $\omega = 2$.

(b) $\omega = 3$.

Figure 1 Our results on the feasibility of achieving CC resilient against t_s corruptions if the network is synchronous and $t_a \leq t_s$ corruptions if it is asynchronous. For a fixed value of $n = 100$, the two plots depict in green the set of pairs (t_s, t_a) for which a protocol exists as percentages of n: the condition $n > \max(\omega \cdot t_s, \omega \cdot t_a + t_s, 2 \cdot t_s + t_a)$. The two black lines correspond to the point-wise optimal resilience thresholds $n > \omega \cdot t_s$ and $n > (\omega + 1) \cdot t_a$ required in the synchronous and asynchronous models respectively. The condition $n > \max(\omega \cdot t_s, \omega \cdot t_a + t_s, 2 \cdot t_s + t_a)$ can be understood as $n > 2 \cdot t_s + t_a$ for $\omega = 2$ and $n > \max(\omega \cdot t_s, \omega \cdot t_a + t_s)$ for $\omega \geq 3$. The two cases are depicted above for $\omega = 2$ and $\omega = 3$.

Feasibility results. Afterwards, we show that the condition $n > \max(\omega \cdot t_s, \omega \cdot t_a + t_s, 2 \cdot t_s + t_a)$ is also sufficient: we give a protocol achieving CC when this condition holds. Together with our impossibility results, this completes the landscape of feasibility for the purely synchronous, purely asynchronous, and network-agnostic models. Our protocol assumes cryptographic setup, namely digital signatures. Note that this is necessary to tolerate $t_s < n/\omega$ corruptions in the synchronous model for $\omega = 2$, a fact which can be easily inferred from various impossibility results (e.g., [23]). When $\omega \geq 3$, however, signatures are no longer required (since $t_s < n/3$ holds in that case), and we will briefly explain how they can be removed. We also note that our protocol is randomized (which is needed [18]), but randomization is restricted to Byzantine Agreement subprotocols [15]. The use of signatures is similarly constrained to Reliable Broadcast and Byzantine Agreement subprotocols [15, 30].

Network-agnostic communication primitives. The core of our CC protocol is a novel network-agnostic implementation of Agreement on a Core-Set (ACS) [8], which may be of independent interest. In essence, ACS allows parties to distribute their inputs and obtain identical views. Our ACS protocol provides stronger guarantees than previous network-agnostic variants [4, 10, 11]. These stronger guarantees will be crucial for achieving CC: when the network is synchronous, we enable the parties to obtain a common view that includes all the honest parties' inputs. Obtaining these properties for ACS when $n > 2 \cdot t_s + t_a$ requires us to diverge from the outline of previous ACS constructions and to instead provide a novel implementation relying on *Gather* (GTHR) [2, 13], a second primitive that we adapt to the network-agnostic model. Roughly, GTHR enables parties to obtain intersecting views.

Prior works corrections. We need to note that our findings seemingly contradict an impossibility result of [33] for general convexity spaces, which depends on the Carathéodory number of the convexity space and not on its Helly number (in general, there is no relationship between the two). Upon closer inspection, we exhibit an error in the proof of [33], meaning that the correct bound is in terms of ω, and not the Carathéodory number.

As a secondary contribution, we identify a core issue in the asynchronous AA protocol for chordal graphs with monophonic convexity of [33]. In the full version of our paper, we describe this issue in detail, and we provide an alternative solution in the network-agnostic model. The alternative protocol is obtained by adapting the AA protocol on cycle-free (chordal) semilattices (i.e., a particular case of chordal graphs) of the same paper [33], while incorporating insights from the protocol of [5] achieving *wait-free* AA on chordal graphs.

1.2 Related work

CC and AA in the pure synchronous and pure asynchronous models. To the best of our knowledge, the problem of agreeing on a value in the honest inputs' convex hull was introduced for AA on $\mathbb{R}$ [16] (where $\omega = 2$). When considering this problem in the synchronous model, $t < n/3$ is tight when no cryptographic setup is allowed [16], and $t < n/2 = n/\omega$ is tight with cryptographic setup [20,25]. In the asynchronous model, the optimal resilience threshold for AA is $t < n/3$ and was proven in [1]. The more general setting of $\mathbb{R}^D$ with straight-line convexity (where $\omega = D + 1$) was first considered in [27,37] (see also the journal version [28]). Here, the bound $t < n/(D + 2)$ is necessary and sufficient for asynchronous AA. Along with the multidimensional variant of AA, Vaidya and Garg [37] have introduced the CC problem on $\mathbb{R}^D$ and showed that the condition $t < n/(D + 1)$ is tight for achieving CC in the synchronous model. Tseng and Vaidya [36] later presented an asynchronous variant of CC resilient against crash failures with incorrect inputs, also on $\mathbb{R}^D$, where parties agree on a polytope in the convex hull of the honest parties' inputs as opposed to a single value.

We also note the works of [19,32], which focus of achieving agreement *on an honest input*. This is a particular case of CC on a space with universe V where a subset's convex hull is the subset itself. Their necessary and sufficient conditions match the more general variants, as the Helly number of this convexity space is $\omega = |V|$: $t < n/|V|$ in the synchronous model [32], and $t < n/(|V| + 1)$ in the asynchronous one [19].

Nowak and Rybicki [33] generalized the problems of CC and AA to abstract convexity spaces. We partially answer an open question raised in [33] on whether there exists an input convexity space for which the optimal resilience threshold for AA depends on the Carathéodory number and not on the Helly number. Our tight conditions for CC imply that, at least for randomized protocols, the asynchronous resilience threshold is actually independent of the Carathéodory number and only depends on the Helly number instead. In addition, we identify a core issue in the deterministic algorithm of [33] for asynchronous AA on chordal graphs. A related line of work considers graph AA in the wait-free model (where $t < n$ of the parties involved may crash) and its variants, primarily focusing on characterizing the families of graphs on which wait-free AA can be achieved [3,5,14,24,26].

CC and AA in the network-agnostic model. The problem of AA on real values has also been considered in the network-agnostic model in [20], where the condition $n > 2 \cdot t_s + t_a$ has been proven necessary and sufficient. For the $\mathbb{R}^D$ variant of AA, the condition $n > (D+1) \cdot t_s + t_a$ has been proven to be sufficient [21], but whether this condition is also necessary is still an open problem. Our work will build upon and extend some of the techniques of [20,21]. Concretely, our Gather protocol is obtained by making an adjustment to the network-agnostic Overlap All-to-All Broadcast primitive of [20,21], while incorporating insights from asynchronous Gather protocols [2,13]. In addition, we rely on similar insights on deriving *safe areas* in the honest parties' convex hulls to [21], and also of prior works in the asynchronous model such as [1,27,33,37]. Previously known techniques and insights will be noted precisely in the following sections. As a summary, our paper will diverge from [20,21] since: (i) we do not assume a particular input space: we consider abstract convexity spaces, and this requires us to provide generalized variants of lower bounds and safe area calculations; (ii) we focus on CC as opposed to AA, and achieving exact agreement requires us to design a stronger communication primitive: ACS. We also need to note that our feasibility result defines the first protocol in the network-agnostic model achieving an optimal resilience trade-off with a non-linear boundary (see Figure 1b). This gives a hint (but not yet an answer) on the question regarding necessary conditions in multidimensional AA left open in [21].

ACS in the network-agnostic model. As previously mentioned, the term ACS has been present in network-agnostic literature, as a building block for State-Machine Replication [4,10] and Multi-Party Computation [11]. We highlight an important distinction between prior constructions and ours. First, prior ACS variants would only provide as output *a set of values*, while our construction provides a common view defined as *a set of value-party pairs*. Second, when running in the synchronous model, the ACS protocols of [4,10,11] only need to ensure that *pre-agreement* is maintained: if all honest parties hold input v, the output set is $\{v\}$. For our CC protocol, the following properties will be crucial: (i) parties agree on the output set if the network is synchronous and t_s of the parties are corrupted (even without pre-agreement); (ii) roughly, each value's multiplicity is reflected in the output set (hence why the output set consists of value-party pairs); and, most importantly, (iii) if the network is synchronous, all honest values are guaranteed to be included (with multiplicities) in the parties' common view. Our ACS implementation will hence focus on this stronger definition, requiring us to diverge from the outline of previous ACS constructions for $n > 2 \cdot t_s + t_a$.

The concurrent work of [22], addressing Atomic Broadcast, also proposes a network-agnostic ACS implementation. While their protocol also relies on Gather and appears to achieve agreement regardless of the type of network, our ACS protocol is strictly stronger, as the protocol proposed by [22] provides the parties with a single value as output, and this value may be proposed by a corrupted party. This would prevent CC, but is sufficient for achieving Atomic Broadcast as in [22], since parties' values are *justified.*

While our ACS definition is stronger than previous network-agnostic variants, the protocol of [4] remains the state-of-art in terms of efficiency. The ACS protocol of [4] achieves an expected communication complexity of $O(n^2 \cdot \ell + n^3 \cdot \kappa)$ bits, where κ is the security parameter, and parties' inputs are represented as ℓ-bit strings (assuming threshold signatures). Even with threshold signatures, our protocol would incur an expected communication complexity of $O(n^3 \cdot \ell + n^4 \cdot \kappa)$, where ℓ denotes the universe elements' size in bits.

2 Preliminaries

In the following, given a non-negative integer k, write $[k]$ for the set $\{1, 2, \ldots, k\}$.

Model. Consider n parties denoted by $P_1, P_2, \ldots, P_n$ running a protocol in a fully-connected network, where links model authenticated channels. A synchronous network ensures that the parties' clocks are perfectly synchronized and that each message is delivered within a publicly known amount of time Δ. If any of these two guarantees fails, then the network is asynchronous. We assume that the parties are not aware a priori of the type of network the protocol is running in. In addition, we assume an adaptive adversary that may corrupt at most t_s parties if the network is synchronous, and at most t_a parties if the network is asynchronous. Corrupted parties permanently become byzantine, meaning that they can deviate arbitrarily, even maliciously, from the protocol. Moreover, the adversary may control the message delivery schedule, subject to the conditions of the network type. We will make use of a public key infrastructure (PKI), and a secure signature scheme. For simplicity, we assume that the signatures are perfectly unforgeable.

Abstract convexity spaces. Given a nonempty set V, also called the *universe*, an *abstract convexity space* on V is a family $\mathcal{C}$ of subsets of V such that $\varnothing, V \in \mathcal{C}$ and $\mathcal{C}$ is closed under arbitrary intersections: whenever $A, B \in \mathcal{C}$, it also holds that $A \cap B \in \mathcal{C}$ (and the infinite analogue). Sets in $\mathcal{C}$ are regarded as *convex sets.* For instance, when $V = \mathbb{R}^D$, one possible $\mathcal{C}$ consists of all sets satisfying the condition that the straight-line segment joining any two points in the set is also included in the set. Note that this yields the standard convexity notion on $\mathbb{R}^D$. However, this is not the only way to define a convexity space on $\mathbb{R}^D$ that is consistent with the definition's requirements; e.g., take $\mathcal{C}$ to be the family of "box" subsets of $\mathbb{R}^D$; i.e., subsets of the form $I_1 \times \ldots \times I_D$, where $(I_i)_{i\in[D]}$ are closed intervals of the real line.

A central notion is that of convex hulls. In particular, the *convex hull* of any (not necessarily convex) set $S \subseteq V$ is the intersection $\langle S\rangle$ of all convex sets $C \in \mathcal{C}$ containing S, which is indeed convex by closure under intersections. In $\mathbb{R}^D$ under straight-line convexity, hulls correspond to the usual notion of Euclidean convex hulls, while under "box"-convexity they correspond to so-called "bounding boxes"; i.e., take the box spanning the region between the infimum and the supremum along each axis. Note that the convex hull operator is idempotent, i.e., $\langle\langle S\rangle\rangle = \langle S\rangle$. Moreover, note that a set is convex if and only if $S = \langle S\rangle$.

The Helly Number ω of a Convexity Space. The following seminal result in convexity theory concerns $\mathbb{R}^D$ with straight-line convexity.

▶ **Theorem 1** (Helly's Theorem). *Consider a finite collection of convex sets in $\mathbb{R}^D$ with straight-line convexity. If every $D + 1$ of them intersect, then all of them intersect.*

Helly's Theorem implies that, for instance, any finite collection of disks in $\mathbb{R}^2$ with triple-wise non-empty intersections has a non-empty intersection. Notice that the same would not hold if $D + 1$ was replaced by D; e.g., one can draw three disks in $\mathbb{R}^2$ that pair-wise intersect but have no point common to all three. One might now wonder: "What about box convexity?" In that case, $D + 1$ can be replaced by 2. For instance, this means that any finite collection of rectangles in $\mathbb{R}^2$ where any two intersect has a non-empty intersection, in contrast to disks. This number, which is $D + 1$ for straight-line convexity and 2 for box convexity, is known as the Helly number ω of the convexity space.

More generally, the *Helly number* ω of a convexity space $\mathcal{C}$ is the smallest number h such that any finite collection of convex sets out of which any h intersect has a non-empty intersection. It is useful to think in terms of the contrapositive: any finite collection of convex sets that do not intersect has a subcollection consisting of (at most) h sets that do not intersect. As a result, ω is equivalently the size of the largest collection of convex sets with an empty intersection such that any of its proper subcollections have a non-empty intersection. Notationally, say that an *m-Helly family* for $\mathcal{C}$ is a collection of m convex sets $C_1, C_2, \ldots, C_m \in \mathcal{C}$ such that their intersection is the empty set, but the intersection of any $m - 1$ of them is non-empty; i.e., $\cap_{j=1}^{m} C_j = \varnothing$ and $\cap_{j\neq i} C_j \neq \varnothing$ for any $i \in [m]$. The *Helly number* ω of $\mathcal{C}$ is then the largest number h such that there exists an h-Helly family for $\mathcal{C}$. We will mostly work with this latter definition of the Helly number. Note that for some spaces, there will exist arbitrarily large Helly families, in which case the Helly number is undefined.[1]

[1] We do not concern ourselves with this case in the statement of our main results, but note that our reasoning often still applies when ω is undefined, for instance when deriving impossibility results. For the rest of this work, we assume that the spaces we consider have a well-defined Helly number ω.

Convex agreement problems. A *convex agreement problem* is defined for a convexity space $\mathcal{C}$ over a universe V; e.g., $\mathbb{R}^D$ with straight-line convexity, or a graph $G = (V, E)$ with monophonic convexity. Each party P starts with an input $v_{\mathsf{in}}^P \in V$ and should produce an output v_{out}^P. Ideally, all outputs should match, and this common output should be in the convex hull of the inputs. However, one has to consider the presence of byzantine parties. Hence, an agreement problem is defined by a collection of properties taking this into account: a validity condition, an agreement condition, and a termination condition. Write V_{in} and V_{out} for the set of inputs v_{in}^P and respectively outputs v_{out}^P of the *honest* parties P.

A convex agreement problem has the following validity and agreement conditions:

Convex Validity: $V_{\mathsf{out}} \subseteq \langle V_{\mathsf{in}} \rangle$ (honest outputs are in the convex hull of honest inputs).
Exact Agreement: $|V_{\mathsf{out}}| = 1$ (honest parties obtain the same output).

Finally, let us discuss the termination requirements of the protocol. There are two flavors, one for deterministic protocols and one for randomized protocols, listed below:

Termination: all honest parties obtain outputs.
Probabilistic Termination: the probability that some honest party has not obtained output after T time units tends to 0 as $T \to \infty$.

We may then define CC as follows:

▶ **Definition 2** (Convex Consensus). *A protocol Π is a (t_s, t_a)-secure CC protocol if it achieves Probabilistic Termination, Convex Validity, and Exact Agreement when up to t_s parties are corrupted if it runs in a synchronous network, and when up to t_a parties are corrupted if running in an asynchronous network.*

3 Resilience Lower Bounds Using the Helly Number

In this section, we establish necessary conditions for achieving CC in the network-agnostic model. Concretely, we show that each of the following conditions is needed: $n > \omega \cdot t_s$, $n > 2 \cdot t_s + t_a$, and $n > \omega \cdot t_a + t_s$. Section 4 will show that these conditions are also sufficient.

We begin by showing that the conditions $n > \omega \cdot t$ and $n > (\omega + 1) \cdot t$ are necessary in the synchronous and resp. asynchronous model, where ω is the Helly number of the convexity space. These already imply that $n > \omega \cdot t_s$ and $n > (\omega + 1) \cdot t_a$ are required in the in the network-agnostic model. Afterward, we move towards conditions that are only required in the network-agnostic model. We note that a previously-known resilience bound [33, Theorem 13] given in terms of the Carathéodory number of the space is incompatible with our results: in general, there is no relation between the Carathéodory number and the Helly number. This bound turns out to be incorrect (detailed discussion in Appendix A).

Before formally showing our lower bounds, we introduce the notion of *adversarial families*, which will enable us to give general scenario-based arguments [32]. Roughly, these are families of pairwise-disjoint sets such that if the honest parties start with inputs from these sets, then Convex Validity forces them to output values from these sets, breaking Exact Agreement. The formal definition follows; see Figure 2 for an example.

▶ **Definition 3.** *Consider a convexity space $\mathcal{C}$ with Helly number ω defined on a universe V. Consider a family $\mathcal{A} = \{A_1, \ldots, A_m\}$ consisting of m non-empty pairwise-disjoint convex sets $A_i \in \mathcal{C}$ and write $A = \cup\mathcal{A}$. Then, $\mathcal{A}$ is m-*adversarial *if $A_i = \cap_{\ell \neq i} \langle A \setminus A_\ell \rangle$ for all $i \in [m]$.*[2]

[2] This definition requires $m > 1$ to avoid taking the intersection of an empty collection of sets. However,

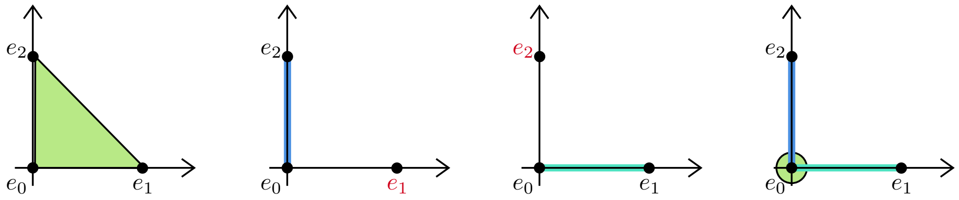


Figure 2 Consider $\mathbb{R}^2$ with straight-line convexity and the vectors $e_0, e_1, e_2 = (0,0), (1,0), (0,1)$, illustrated in the first figure. Define $A_i = \{e_i\}$ for $i = 0, 1, 2$ to be singleton sets for the previous (and hence convex sets). Then, $\mathcal{A} = \{A_0, A_1, A_2\}$ is a 3-adversarial family. To see why, first note that by definition $A = \{e_0, e_1, e_2\}$. By symmetry, it suffices to check the condition for $i = 0$; i.e., to show that $A_0 = \langle A \setminus A_1\rangle \cap \langle A \setminus A_2\rangle$. Simplifying, this amounts to $A_0 = \langle\{e_0, e_2\}\rangle \cap \langle\{e_0, e_1\}\rangle$. The second figure illustrates $\langle\{e_0, e_2\}\rangle$, the third illustrates $\langle\{e_0, e_1\}\rangle$, and the forth $\langle\{e_0, e_2\}\rangle \cap \langle\{e_0, e_1\}\rangle$, which is precisely A_0, as required. The illustrations come from [21].

Note that the pairwise-disjoint condition is equivalent to $\cap_{\ell=1}^{m}\langle A \setminus A_\ell\rangle = \varnothing$.[3] We add that, in the example of Figure 2, sets A_i are singletons, but requiring this would strictly decrease the power of adversarial sets in general. The following technical lemma, and the two following it, will be the main tools used to get impossibility results. The techniques used in its proof, supplied in the full version of our paper, are similar in spirit to the proofs for $\mathbb{R}^D$ in [28].

▶ **Lemma 4.** *Let $\mathcal{A} = \{A_1, \ldots, A_m\}$ be an m-adversarial family for convexity space $\mathcal{C}$. Assume $n \geq m$ and that, moreover, $n \leq m \cdot t$ if the network is synchronous and $n \leq (m+1) \cdot t$ if the network is asynchronous. Then, any n-party protocol satisfying Convex Validity and (Probabilistic) Termination has a terminating execution where there are honest parties $P_1, \ldots, P_m$ such that the output v^i_{out} of party P_i satisfies $v^i_{\mathsf{out}} \in A_i$.*

The following two technical lemmas give similar guarantees, but in the network-agnostic model. The proof of the first is similar to that for $\mathbb{R}$ in [20], while that of the second is an extension of the asynchronous part of Lemma 4. The proofs are included in the full version of our paper.

▶ **Lemma 5.** *Assume a convexity space $\mathcal{C}$ admitting a 2-adversarial family $\mathcal{A} = \{A_1, A_2\}$. Assume $2 \leq n \leq 2 \cdot t_s + t_a$. Let Π denote an arbitrary protocol achieving Convex Validity and (Probabilistic) Termination for at most t_s corruptions when the network is synchronous and at most t_a corruptions when it is asynchronous. Then, Π has a terminating execution where the outputs v^1_{out} and v^2_{out} of two honest parties satisfy $v^1_{\mathsf{out}} \in A_1$ and $v^2_{\mathsf{out}} \in A_2$.*

▶ **Lemma 6.** *Let $\mathcal{A} = \{A_1, \ldots, A_m\}$ be an m-adversarial family for convexity space $\mathcal{C}$. Assume that $m \leq n \leq m \cdot t_a + t_s$. Then, any n-party protocol satisfying Convex Validity and (Probabilistic) Termination for at most t_s corruptions when the network is synchronous and at most t_a corruptions when the network is asynchronous has a terminating execution where there are honest parties $P_1, \ldots, P_m$ such that the output v^i_{out} of party P_i satisfies $v^i_{\mathsf{out}} \in A_i$.*

We now show a relationship between adversarial families and Helly families.

▶ **Lemma 7.** *Consider a convexity space $\mathcal{C}$, then an m-adversarial family exists if and only if an m-Helly family exists. Hence, the size of the largest adversarial family for a convexity space equals its Helly number ω.*

for $m = 1$ all our results will hold if we assume that $\mathcal{A} = \{A\}$ is 1-adversarial for any convex set $A \neq \varnothing$. We will not discuss this technicality further and henceforth assume that $m \geq 1$ is well-defined.

[3] To see this, note that for $i \neq j$ we have $A_i \cap A_j = (\cap_{\ell \neq i}\langle A \setminus A_\ell\rangle) \cap (\cap_{\ell \neq j}\langle A \setminus A_\ell\rangle) = \cap_{\ell=1}^{m}\langle A \setminus A_\ell\rangle$.

Proof. First, consider an adversarial family $\mathcal{A} = \{A_1, \dots, A_m\}$ for $\mathcal{C}$ and as usual write $A = \cup\mathcal{A}$. The family of sets $\langle A \setminus A_i \rangle_{i \in [m]}$ do not intersect, but any $m-1$ of them do, since for any i we assumed that $A_i = \cap_{\ell \neq i} \langle A \setminus A_\ell \rangle$ is non-empty, so it is an m-Helly family. Conversely, consider an m-Helly family; i.e., convex sets $C_1, \dots, C_m \in \mathcal{C}$ that do not intersect, but any $m-1$ of them do. Define the family of non-empty convex sets $\mathcal{A} = \{A_1, \dots, A_m\}$ where $A_i = \cap_{\ell \neq i} C_\ell$. Notice that for $i \neq j$ we have $A_i \cap A_j = \cap_{\ell \in [m]} C_\ell = \varnothing$, so the sets are pairwise disjoint. To show that $\mathcal{A}$ is an m-adversarial family, it remains to show that for all i it holds that $A_i = \cap_{\ell \neq i} \langle A \setminus A_\ell \rangle$. To see this, note that $A \setminus A_\ell = \cup\{A_1, \dots, A_{\ell-1}, A_{\ell+1}, \dots, A_m\}$ and that $A_{\ell'} \subseteq C_\ell$ for all $\ell' \neq \ell$, so $A \setminus A_\ell \subseteq C_\ell$. Since C_ℓ is convex, this means that $\langle A \setminus A_\ell \rangle \subseteq \langle C_\ell \rangle = C_\ell$. As a result, $\cap_{\ell \neq i} \langle A \setminus A_\ell \rangle \subseteq \cap_{\ell \neq i} C_\ell = A_i$. To also show that $A_i \subseteq \cap_{\ell \neq i} \langle A \setminus A_\ell \rangle$ just notice that $A_i \subseteq A \setminus A_\ell \subseteq \langle A \setminus A_\ell \rangle$ for all $\ell \neq i$. ◀

Note that a more restrictive definition of adversarial families where all the sets are singletons would not suffice to prove the previous, as in some spaces no singletons are convex.

To access the full power of Lemmas 4 and 6, which require n to be at least the size of the adversarial family, we would like that adversarial families of a certain size imply the existence of adversarial families of all smaller sizes. We show this in the following lemma.

▶ **Lemma 8.** *Given a convexity space, if there exists an m-Helly family, then there exist m'-Helly families for any $1 \leq m' < m$. The same holds if "Helly" is replaced by "adversarial."*

Proof. It suffices to consider $m' = m - 1$. If $C_1, \dots, C_m$ is an m-Helly family, one can check that $C_1, \dots, C_{m-2}, (C_{m-1} \cap C_m)$ is an $(m-1)$-Helly family. For the latter, apply Lemma 7. ◀

We now leverage Lemmas 4, 7 and 8 to get the following result, generalizing those in [33] by removing the strong requirement of a convex geometry.

▶ **Theorem 9.** *Consider a convexity space $\mathcal{C}$ with Helly number ω. Assume $n \leq \omega \cdot t$ if the network is synchronous and $n \leq (\omega + 1) \cdot t$ if the network is asynchronous. Then, there is no n-party protocol satisfying Convex Validity and (Probabilistic) Termination such that the set of outputs of the honest parties is guaranteed to have size at most $\min(n, \omega) - 1$.*

Proof. Write $m = \min(n, \omega)$. By Lemma 7, there is an ω-adversarial family for $\mathcal{C}$. Since $m \leq \omega$, using Lemma 8, let $\mathcal{A} = \{A_1, \dots, A_m\}$ be an m-adversarial family for $\mathcal{C}$. Consider a protocol Π satisfying Convex Validity and Termination. By Lemma 4, there is a terminating execution of Π where the set of honest outputs contains $\{a_1, \dots, a_m\}$ where $a_i \in A_i$. As sets in $\mathcal{A}$ are pairwise disjoint, this set has cardinality m, implying the conclusion. ◀

By leveraging Lemmas 5, 6, 7 and 8, we similarly get the following result. The proof is included in the full version of our paper.

▶ **Theorem 10.** *Consider a convexity space $\mathcal{C}$ with Helly number $\omega \geq 2$. Assume $2 \leq n \leq 2 \cdot t_s + t_a$ or $2 \leq n \leq \omega \cdot t_a + t_s$. Then, no n-party protocol satisfying Convex Validity, (Probabilistic) Termination, and Exact Agreement can simultaneously tolerate at most t_s corruptions when the network is synchronous and at most t_a when the network is asynchronous.*

4 Achieving Optimal-Resilience Convex Consensus

We now describe a construction achieving CC in the network-agnostic model that matches our previous resilience lower bounds. Concretely, we focus on proving the following theorem.

▶ **Theorem 11.** *If $t_a \leq t_s$ and $n > \max(\omega \cdot t_s, 2 \cdot t_s + t_a, \omega \cdot t_a + t_s)$, there is a protocol achieving (t_s, t_a)-secure* CC *assuming PKI. The protocol has expected round complexity $O(1)$. If ℓ denotes the universe elements' size in bits and κ is the security parameter, its expected communication complexity is $O(n^3 \cdot \ell + n^4 \cdot \kappa)$ bits. If threshold signatures are available, the expected communication complexity reduces to $O(n^3 \cdot \ell + n^3 \cdot \kappa)$ bits.*

To set up the intuition for our construction, we recall the outline of the synchronous protocol for $\mathbb{R}^D$ with straight-line convexity of [37]. The synchronous model offers powerful communication primitives (i.e., Synchronous Broadcast [17]), enabling the parties to distribute their inputs and obtain an *identical view* of the inputs. This view consists of a set of value-sender pairs, out of which $n - t_s$ correspond to honest parties. Then, the parties derive a *safe area* inside the honest inputs' convex hull by intersecting the convex hulls of all subsets of $n - t_s$ values received, as defined below. We extend the convex hull operator to value-sender sets straightforwardly: ignore party identities and take the convex hull of the values.

▶ **Definition 12** (Safe Area). *Let $\mathcal{M}$ denote a set of value-sender pairs. For a given k, $safe_k(\mathcal{M}) := \bigcap_{M \in restrict_k(\mathcal{M})} \langle M \rangle$, where $restrict_k(\mathcal{M}) := \{M \subseteq \mathcal{M} : |M| = |\mathcal{M}| - k\}$.*

Specifically, if parties received the (same) set $\mathcal{M}$ of $n - t_s + k$ value-sender pairs, they compute their safe area as $safe_k(\mathcal{M})$. We will later show (in a more general form) that, since $n > \omega \cdot t_s$, the safe area obtained is non-empty. Therefore, any value in the common safe area is valid. Hence, parties may output any such value chosen by some deterministic criterion.

Technical assumptions. Implementing such a protocol requires mild assumptions about $\mathcal{C}$: it should be possible to (i) store elements from V and to send them in messages; (ii) compute and intersect convex hulls; (iii) deterministically select a point from the safe area. *Only to express communication complexity bounds, we will also need that $|V| \leq 2^\ell$ for some ℓ.*

Identical views in asynchrony. Building towards our solution achieving network-agnostic guarantees, we first identify the challenges posed by translating the outline above to the purely asynchronous model (where $t_s = t_a$ and $n > (\omega + 1) \cdot t_a$). Given a primitive that provides the parties with an identical view of $n - t_a$ value-sender pairs, CC can be achieved in a similar manner: out of the set $\mathcal{M}$ of $n - t_a$ pairs agreed upon, at most t_a are corrupted. Then, honest parties derive the safe area $safe_{t_a}(\mathcal{M})$ inside their inputs' convex hull, and afterwards take a deterministic decision to obtain the same output.

Achieving the required identical view deterministically is impossible in the asynchronous model [18], but randomization allows for a simple solution by employing a primitive introduced in [7]. This primitive archives *Agreement on a Core-Set* (ACS) when up to $t_a < n/3$ of the parties involved are corrupted, which suffices for our case of $\omega \geq 2$. Roughly speaking, an ACS protocol assumes that each party holds a value meant to be distributed, and enables the parties to obtain the same set $\mathcal{M}$ of $n - t_a$ value-sender pairs. By utilizing the (randomized) ACS protocol presented in [8, Section 4], we achieve asynchronous CC with optimal resilience, in constant expected number of rounds, proving the lower bound $n > (\omega + 1) \cdot t_a$ to be tight.

Exploiting the advantages of synchrony. While the standard definition of ACS provides identical views when the network is asynchronous, the synchronous model still has a crucial advantage that needs to be used to achieve higher resilience. Namely, the key insight on why CC can be achieved up to $t_s < n/\omega$ corruptions in the synchronous model, while $t_a < n/(\omega + 1)$ is necessary in the asynchronous one, is that the former ensures all honest values are delivered. In contrast, in the asynchronous setting, t_a corrupted parties may *replace* t_a honest parties: the honest parties' messages get delayed for sufficiently long, while the t_a corrupted parties follow the protocol correctly, but with inputs of their choice. To match the condition $n > \max(\omega \cdot t_s, 2 \cdot t_s + t_a, \omega \cdot t_a + t_s)$ in the network-agnostic model, we hence need an additional property in a synchronous network: all honest values must be included in the output set. Consequently, we propose the following enriched definition:

▶ **Definition 13** (Agreement on a Core-Set). *Let Π be a protocol where every party P holds an input v_P and outputs a set of value-sender pairs $\mathcal{M}_P$. We consider the following properties:*
***Validity**: Let P and P' be two honest parties. If $(v', P') \in \mathcal{M}_P$, then $v' = v_{P'}$.*
***Consistency**: If P and P' are honest, $(v, P'') \in \mathcal{M}_P$ and $(v', P'') \in \mathcal{M}_{P'}$, then $v = v'$.*[4]
***T-Output Size**: If an honest party P outputs $\mathcal{M}_P$, then $|\mathcal{M}_P| \geq n - T$.*
***Honest Core**: If an honest party P outputs $\mathcal{M}_P$, then $(v_{P'}, P') \in \mathcal{M}_P$ for every honest P'.*

Then, we say that Π is a (t_s, t_a)-secure ACS *protocol if it achieves the following:*

- *Validity, Consistency, Exact Agreement, Honest Core, Probabilistic Termination when running in a synchronous network where at most t_s parties are corrupted;*
- *Validity, Consistency, Exact Agreement, t_s-Output Size,*[5] *Probabilistic Termination when running in an asynchronous network where at most t_a parties are corrupted.*

Section 5 describes a protocol Π_{ACS} realizing the theorem below (given PKI). We will also describe a protocol for $t_a \leq t_s < n/3$ without PKI, which is suitable for the case $\omega \geq 3$.

▶ **Theorem 14.** *If $n > 2 \cdot t_s + t_a$ and $t_a \leq t_s$, there is a (t_s, t_a)-secure* ACS *protocol Π_{ACS} (assuming PKI). The protocol has expected round complexity $O(1)$. If ℓ denotes the universe elements' size in bits and κ is the security parameter, its expected communication complexity is $O(n^3 \cdot \ell + n^4 \cdot \kappa)$ bits. If threshold signatures are available, the expected communication complexity reduces to bits.*

If parties distribute their values via Π_{ACS}, they agree on a set $\mathcal{M}$ of $n - t_s + k$ value-sender pairs. If the network is asynchronous, at most t_a of these values are corrupted. In contrast, if the network is synchronous, at most k of these values are corrupted due to Honest Core. To cover both cases, parties locally compute their safe areas as $S := \mathit{safe}_{\max(k, t_a)}(\mathcal{M})$ and deterministically decide on an output $v_{\mathsf{out}} \in S$. Note that, by definition, S is indeed inside the honest inputs' convex hull. In addition, since the input space has Helly number ω and $n > \max(\omega \cdot t_s, \omega \cdot t_a + t_s)$, the safe area can be shown to be non-empty, so such v_{out} can be chosen. The proof of this result, stated below, is contained in the full version of our paper.

▶ **Lemma 15.** *Assume $n > \max(\omega \cdot t_s, \omega \cdot t_a + t_s)$, and that $\mathcal{M}$ is a set of $n - t_s + k$ value-party pairs, where $0 \leq k \leq t_s$. Then, $\mathit{safe}_{\max(k, t_a)}(\mathcal{M}) \neq \varnothing$.*

We may now conclude the section by providing the formal code of our CC protocol.

Protocol Π_{CC}

Code for party P with input v_{in}

```
Join $\Pi_{\mathsf{ACS}}$ with input $v_{\mathsf{in}}$. Upon obtaining output $\mathcal{M}$:
    $k := |\mathcal{M}| - (n - t_s)$; $S := \mathit{safe}_{\max(k, t_a)}(\mathcal{M})$.
    Choose $v_{\mathsf{out}} \in S$ according to a public, predetermined, deterministic rule. Output $v_{\mathsf{out}}$.
```

[4] Note that this implies that each party appears at most once as a sender in $\mathcal{M}_P$.
[5] This is intentional: we do not require the stronger property of t_a-Output Size.

Proof of Theorem 11. Π_{ACS} provides the parties with the same set $\mathcal{M}$ of $n - t_s + k$ value-sender pairs, with $0 \leq k \leq t_s$. $\mathcal{M}$ contains at most $\max(k, t_a)$ values from byzantine parties: in a synchronous network, this holds due to Π_{ACS}'s Honest Core property. Hence, there is a subset $M_H \subseteq \mathcal{M}$ of size $|\mathcal{M}| - \max(k, t_a)$ only containing honest inputs. By definition, $M_H \in \mathit{restrict}_{\max(k,t_a)}(\mathcal{M})$, so $S \subseteq \langle M_H \rangle$, i.e., honest parties obtain a safe area S that is included in their inputs' convex hull. Lemma 15 ensures that S is non-empty, and therefore honest parties agree on the same value v_{out} in the honest inputs' convex hull. Consequently, Π_{CC} is (t_s, t_a)-secure CC protocol. ◀

5 Agreement on a Core-Set

In this section, we describe the protocol realizing Theorem 14. We first focus on the easier case $t_a \leq t_s < n/3$: we begin by describing the asynchronous protocol of [8] (as presented in [29]), which fulfills all properties outlined in Definition 13, except for Honest Core in a synchronous network. We adapt this protocol to satisfy Honest Core as well, obtaining (t_s, t_a)-secure ACS when $t_a \leq t_s < n/3$. Note that our CC lower bound of $n > \max(\omega \cdot t_s, 2 \cdot t_s + t_a, \omega \cdot t_a + t_s)$ implies $t_a \leq t_s < n/3$ for $\omega \geq 3$, so the adapted protocol suffices if the latter is true. For $\omega = 2$, however, the adapted protocol is insufficient. Consequently, we then move on to the case $n > 2 \cdot t_s + t_a$, for which we present a novel construction.

We will utilize Reliable Broadcast (rBC) and Byzantine Agreement (BA) as building blocks. We include their definitions in the network-agnostic model below.

▶ **Definition 16** (Reliable Broadcast). *Let Π denote a protocol where a designated party S (the sender) holds a value v_S, and every party P may a value output v_P. Consider the following properties:*

***Validity**: If S is honest, and an honest party outputs v_P, then $v_P = v_S$.*

***Consistency**: If P and P' are honest and output v_P and resp. $v_{P'}$, then $v_P = v_{P'}$.*

***Honest Termination**: If S is honest, all honest parties obtain outputs.*

***Conditional Termination**: If an honest party P outputs, all honest parties obtain outputs.*

We say that Π is a (t_s, t_a)-secure rBC protocol if it achieves Validity, Consistency, Honest Termination, and Conditional Termination up to t_s corruptions if it runs in a synchronous network, and up to t_a corruptions if it runs in an asynchronous network.

▶ **Definition 17** (Byzantine Agreement). *Let Π be a protocol where every party P holds a bit as input and may output a bit, and consider the following property:*

***Weak Validity**: If all honest parties hold input b, no honest party outputs $b' \neq b$.*

Then, Π is a (t_s, t_a)-secure BA protocol if it achieves Weak Validity, Exact Agreement, and Probabilistic Termination up to t_s corruptions if it runs in a synchronous network, and up to t_a corruptions if it runs in an asynchronous network.

The asynchronous ACS protocol of [8]. We describe the protocol of [8], following the variant presented in [29]. We denote this protocol by $\Pi_{a\mathsf{ACS}}$. We highlight once again that $\Pi_{a\mathsf{ACS}}$ is designed for the purely asynchronous model: it assumes a single threshold $t < n/3$ and seeks properties that hold under asynchrony with at most t corrupted parties. To use this protocol in the hybrid model with $t_a \leq t_s < n/3$, one can set $t := t_s$. When this is done, $\Pi_{a\mathsf{ACS}}$ achieves all properties of being a (t_s, t_a)-secure ACS protocol as per Definition 13 (in fact, even (t_s, t_s)-secure), with the exception of Honest Core under synchrony.

In $\Pi_{a\mathsf{ACS}}$, every party first distributes its input value via rBC. Concretely, this is done using Bracha's protocol [12], denoted by $\Pi_{a\mathsf{rBC}}$, which achieves (t, t)-secure rBC for $t < n/3$. Due to asynchronous communication, $\Pi_{a\mathsf{rBC}}$ only guarantees that parties receive a value if the sender is honest. As a result, at least $n-t$ values eventually get delivered to all parties, but these values might still be received at vastly different times, leading to inconsistent views if parties were to output the first $n-t$ values they received.

Then, to decide on a common output set, the parties will use BA to agree on which values they received and should be included in the output set. We utilize the protocol $\Pi_{a\mathsf{BA}}$ of Mostefaoui et al. [31], which achieves (t,t)-secure BA when $t < n/3$. There will be n parallel invocations of $\Pi_{a\mathsf{BA}}$ – one for each party. When a party P receives a value v from P' via $\Pi_{ar\mathsf{BC}}$, it joins the $\Pi_{a\mathsf{BA}}$ invocation corresponding to P' with input 1. Semantically, if the $\Pi_{a\mathsf{BA}}$ invocation of a party P' returns output 1, then the value distributed by P' via $\Pi_{ar\mathsf{BC}}$ should be included in the output set. Note that, when this happens, the Weak Validity property of $\Pi_{a\mathsf{BA}}$ ensures that at least one honest party P has joined the $\Pi_{a\mathsf{BA}}$ invocation for party P' with input 1. That is, P has received a value v from P', and $\Pi_{ar\mathsf{BC}}$'s Conditional Termination ensures that all honest parties eventually receive the same value v from P'. In simple terms, the value of P' is *worth waiting for*, and parties wait until they receive it before completing the protocol.

Eventually, at least $n-t$ invocations result in output 1 (suppose not, then, since at least $n-t$ honest values are eventually delivered to all honest parties, the honest parties will all join at least $n-t$ invocations of $\Pi_{a\mathsf{BA}}$ with input 1, guaranteeing that those invocations terminate with output 1). To complete the protocol, we still need that all $\Pi_{a\mathsf{BA}}$ invocations complete. Then, whenever some party P observes $n-t$ invocations completing with output 1, it should join all remaining invocations with input 0. Hence, once all honest parties join all $\Pi_{a\mathsf{BA}}$ invocations, all invocations are guaranteed to terminate. Upon observing that all $\Pi_{a\mathsf{BA}}$ invocations have terminated, each party P outputs the set of (at least $n-t$) value-sender pairs corresponding to the $\Pi_{a\mathsf{BA}}$ invocations that returned output 1 (after waiting to receive any outstanding values through $\Pi_{ar\mathsf{BC}}$). This way, the protocol ensures that all honest parties obtain an identical view, achieving all properties required by Definition 13 for asynchronous networks. Note that the Validity property follows immediately from the guarantees of $\Pi_{ar\mathsf{BC}}$.

Adjustments for the network-agnostic model. We now return to the network-agnostic model and adapt $\Pi_{a\mathsf{ACS}}$ to achieve our ACS definition for the parameter range $t_a \le t_s < n/3$. As previously established, $\Pi_{a\mathsf{ACS}}$ already satisfies all properties required to be a (t_s, t_a)-secure ACS protocol by setting $t := t_s$, except for Honest Core in a synchronous network.

To see why the Honest Core property does not hold, consider a scenario where the network is synchronous, and the t_s corrupted parties follow the protocol correctly with inputs of their choice. All messages are delivered immediately, except for the messages sent by t_s of the honest parties: these are delivered exactly after Δ time. The remaining honest parties complete the protocol before time Δ, and the values of t_s honest parties are missing from the output set. To prevent this, we impose a waiting time to ensure that, if the network is synchronous, all honest parties' messages are received. Running $\Pi_{ar\mathsf{BC}}$ in the synchronous model guarantees additional properties, established in [21]: when an honest party sends a value via $\Pi_{ar\mathsf{BC}}$, all parties receive this output within $c_{ar\mathsf{BC}} \cdot \Delta$ time, where $c_{ar\mathsf{BC}} := 3$. Then, to achieve Honest Core, we impose a waiting period of at least $c_{ar\mathsf{BC}} \cdot \Delta$ time before allowing the parties to participate in $\Pi_{a\mathsf{BA}}$ invocations with input 0. This way, if the network is synchronous, all honest parties join every honest party's $\Pi_{a\mathsf{BA}}$ invocation with input 1, and these invocations return 1. Hence, all honest parties' values are included in the output set.

We include below the adapted $\Pi_{a\mathsf{ACS}}$, which achieves (t_s, t_a)-secure ACS for $t_a \le t_s < n/3$. In fact, it achieves (t_s, t_s)-secure ACS: it is an asynchronous protocol with an added waiting period to ensure the Honest Core property under synchrony. The protocol incurs expected constant round complexity, and expected communication complexity $O(n^3 \cdot \ell)$, where ℓ denotes the universe elements' size. For the formal analysis, see the full version of our paper.

Adapted protocol $\Pi_{a\mathsf{ACS}}$ [8, 29]

Code for party P with input v ($\Pi_{ar\mathsf{BC}}$ and $\Pi_{a\mathsf{BA}}$ invocations use $t := t_s$)

```
$\tau_{\texttt{start}} := \tau_{\texttt{now}}$
Send $v$ to every party via $\Pi_{ar\mathsf{BC}}$.
When receiving a value $v$ from $P'$ via $\Pi_{ar\mathsf{BC}}$:
    If $\tau_{\texttt{now}} \leq \tau_{\texttt{start}} + c_{ar\mathsf{BC}} \cdot \Delta$ or less than $n - t_s$ invocations of $\Pi_{ar\mathsf{BC}}$ returned 1:
        Join the invocation of $\Pi_{a\mathsf{BA}}$ for $P'$ with input 1.
When $\tau_{\texttt{now}} > \tau_{\texttt{start}} + c_{ar\mathsf{BC}} \cdot \Delta$ and at least $n - t_s$ of the $\Pi_{a\mathsf{BA}}$ invocations returned 1:
    Join the remaining $\Pi_{a\mathsf{BA}}$ invocations with input 0.
When all $\Pi_{a\mathsf{BA}}$ invocations have terminated:
    $\mathcal{P}$ := parties whose corresponding $\Pi_{a\mathsf{BA}}$ invocations have terminated with output 1.
    When all invocations of $\Pi_{ar\mathsf{BC}}$ having senders in $\mathcal{P}$ have terminated:
        $\mathcal{M}$ := the set of pairs $(v', P')$, where $P' \in \mathcal{P}$ and $v'$ is the value $P'$ sent via $\Pi_{ar\mathsf{BC}}$.
        Output $\mathcal{M}$.
```

Achieving ACS when $n > 2 \cdot t_s + t_a$. Finally, we describe our solution for the general case. We utilize building blocks designed specifically for this setting: the (t_s, t_a)-secure rBC protocol Π_{rBC} of Momose and Ren [30], and the (t_s, t_a)-secure BA protocol Π_{BA} of Deligios, Hirt and Liu-Zhang [15, Corollary 2]. We add that these protocols assume and need PKI.

While the $t_a \leq t_s < n/3$ setting enabled a solution based on tweaks to previously known protocols, the $n > 2 \cdot t_s + t_a$ case introduces different challenges. In particular, one detail we omitted when presenting $\Pi_{a\mathsf{ACS}}$ concerns protocol composability. Namely, Definition 17 of BA protocols assumes that honest parties join the protocol simultaneously in a synchronous network. In the outline of [8] and in $\Pi_{a\mathsf{ACS}}$, this is, in fact, not the case. However, when $t_s = t_a$, this assumption is not strictly required because the asynchronous guarantees step in whenever honest parties are unable to join simultaneously. In contrast, when $t_s > t_a$, (t_s, t_a)-secure BA does not have to provide any guarantees in a synchronous network with t_s corruptions if honest parties are unable to join simultaneously. This is especially problematic when $t_s \geq n/3$ because (t_s, t_s)-secure BA protocols do not exist. The Conditional Termination property of Π_{rBC} is too weak to ensure that honest parties are ready to join the BA invocations simultaneously when the network is synchronous – a challenge that we need to overcome.

Our goal is then to allow the parties to join each invocation of Π_{BA} at the same time when the network is synchronous – this refers both to invocations where parties join with input 1, and to invocations where parties join with input 0. We will do so by enabling the parties to decide their input bit for each invocation of Π_{BA} independently of the other invocations' outcomes. On top of this property, we still need to guarantee t_s-Output Size when the network is asynchronous, which amounts to ensuring that at least $n - t_s$ invocations of Π_{BA} return 1. Moreover, when the network is synchronous, the Π_{BA} invocations of honest parties have to output 1 to ensure the Honest Core property.

To enable the honest parties to safely decide each input bit for the Π_{BA} invocations independently, realizing the previous desiderata, we introduce a network-agnostic version of a primitive known as *Gather* (GTHR) [2,13]. This is a slightly weaker, but deterministic variant of ACS: GTHR relaxes Exact Agreement by only requiring that honest parties' output sets have at least $n - t_s$ values in common. Our definition of GTHR, provided below, additionally requires the previous Honest Core property to hold under synchrony. Moreover, we require that honest parties obtain outputs simultaneously if the network is synchronous.

▶ **Definition 18** (Gather). *Let Π be a protocol where every party P holds an input v_P and may output a set of value-sender pairs $\mathcal{M}_P$. We consider the following properties, additionally to those in Definition 13.*

T-Common Core: *If all honest parties obtain outputs, then* $\left|\bigcap_{P \text{ honest}} \mathcal{M}_P\right| \geq n - T$.
Simultaneous Termination: *The honest parties obtain outputs simultaneously.*

Then, we say that Π *is a* (t_s, t_a)*-secure* GTHR *protocol if it achieves:*

- *Validity, Consistency, Honest Core and Simultaneous Termination when running in a synchronous setting where at most* t_s *of the parties involved are corrupted;*
- *Validity, Consistency,* t_s*-Common Core and Termination when running in an asynchronous setting where at most* t_a *of the parties involved are corrupted.*

In Appendix B, we provide a construction achieving our GTHR definition, as stated below. This is obtained by adding one step to the network-agnostic Overlap All-to-All Broadcast protocol of [20], while using insights from asynchronous variants of GTHR [2].

▶ **Theorem 19.** *If* $n > 2 \cdot t_s + t_a$ *and* $t_a \leq t_s$*, there is a* (t_s, t_a)*-secure* GTHR *protocol* Π_{GTHR} *(assuming PKI). The protocol has round complexity* $O(1)$*. If* ℓ *denotes the universe elements' size in bits and* κ *is the security parameter, it achieves a communication complexity of* $O(n^3 \cdot \ell + n^4 \cdot \kappa)$ *bits (can be reduced to* $O(n^3 \cdot \ell + n^3 \cdot \kappa)$ *with threshold signatures).*

Then, our ACS protocol proceeds as follows: the parties distribute their inputs using the Π_{GTHR} protocol realizing Theorem 19. When obtaining outputs $\mathcal{M}_{\mathsf{GTHR}}$ (potentially different for different parties), each party P joins the n invocations of Π_{BA}. P inputs 1 in the invocation for P' if $\mathcal{M}_{\mathsf{GTHR}}$ contains some value from P' and 0 otherwise. Note that, if the network is synchronous, Π_{GTHR} provides Simultaneous Termination, hence honest parties join Π_{BA} simultaneously, and therefore the guarantees of Π_{BA} now hold. Since the value-sender pairs $\mathcal{M}_{\mathsf{GTHR}}$ obtained by the honest parties intersect in $n - t_s$ pairs, at least $n - t_s$ of the Π_{BA} invocations result in output 1. In addition, if the network is synchronous, Π_{GTHR}'s Honest Core ensures that every invocation corresponding to an honest party returns 1.

An important and subtle caveat in the above is that obtaining output 1 in the Π_{BA} invocation for P' does not mean that all honest parties have received a value from P' via Π_{GTHR}. Although Π_{BA} ensures that at least one honest party P has received a value from P' via Π_{GTHR}, this may not be the case for all honest parties. To address this, we state an additional property provided by our implementation of Π_{GTHR}: if parties wait for sufficiently long even after obtaining outputs via Π_{GTHR}, they will (consistently) receive the missing values as well. This is because, internally to Π_{GTHR}, parties distribute their inputs via Π_{rBC}.

▶ **Observation 20.** *Let* P *and* P' *denote two honest parties, and let* $\mathcal{M}$ *and* $\mathcal{M}'$ *denote their internal message sets in* Π_{GTHR}*. If* $(v, P'') \in \mathcal{M}$*, then, eventually,* $(v, P'') \in \mathcal{M}'$ *as well.*

We may now present our ACS protocol. We defer the analysis to Appendix C.

Protocol Π_{ACS}

Code for party P with input v

```
Join $\Pi_{\mathsf{GTHR}}$ with input $v$. When receiving output $\mathcal{M}_{\mathsf{GTHR}}$ from $\Pi_{\mathsf{GTHR}}$:
    For each party $P'$:
        $b_{P'} := 1$ if $(v', P') \in \mathcal{M}_{\mathsf{GTHR}}$ for some $v'$ and 0 otherwise.
        Join the $\Pi_{\mathsf{BA}}$ invocation for $P'$ with input $b_{P'}$.
    When receiving $v'$ from $P'$ via $\Pi_{\mathsf{rBC}}$ [initiated in $\Pi_{\mathsf{GTHR}}$], add $(v', P')$ to $\mathcal{M}_{\mathsf{GTHR}}$.
    When obtaining outputs in all invocations of $\Pi_{\mathsf{BA}}$:
        $\mathcal{P} :=$ the set of parties whose $\Pi_{\mathsf{BA}}$ invocations returned output 1.
        When $(v', P') \in \mathcal{M}_{\mathsf{GTHR}}$ for every $P' \in \mathcal{P}$:
            Output $\mathcal{M} :=$ the set of pairs $(v', P') \in \mathcal{M}_{\mathsf{GTHR}}$ with $P' \in \mathcal{P}$.
```

6 Conclusions

We investigated the necessary and sufficient conditions for achieving CC in the network-agnostic model, providing a necessary and sufficient condition for solvability. We have seen that, for any convexity space with Helly number ω, achieving CC, or, more precisely, Convex Hull Validity, (Probabilistic) Termination and Agreement on at most $\min(n, \omega) - 1$ values requires $n > \omega \cdot t$ in synchronous networks and $n > (\omega + 1) \cdot t$ in asynchronous networks. In the network-agnostic model, we have shown that $n > \max(\omega \cdot t_s, \omega \cdot t_a + t_s, 2 \cdot t_s + t_a)$ is necessary and sufficient for achieving CC. To this end, we provided a (t_s, t_a)-secure CC protocol Π_{CC} by making use of randomization, which can be seen to be necessary due to the FLP result [18], and assuming PKI only for the particular case $\omega = 2$ (where cryptographic setup is needed when $t_s \geq n/3$ [23]).

In the process, we proposed two communication primitives for the network-agnostic model, which may be of independent interest. These are variants of ACS and GTHR, which allow each party to distribute its input so that parties obtain consistent views on the original inputs. These stronger properties enabled us to ensure the synchronous resilience guarantees of CC in the network-agnostic model. With its stronger guarantees, our ACS protocol can simplify future works on network-agnostic secure Multi-Party Computation, where ACS protocols are often employed during the input-sharing part of the protocol (for instance, [11] uses a less general form of ACS).

References

1 Ittai Abraham, Yonatan Amit, and Danny Dolev. Optimal resilience asynchronous approximate agreement. In Teruo Higashino, editor, *Principles of Distributed Systems*, pages 229–239, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

2 Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 363–373, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467914.

3 Manuel Alcántara, Armando Castañeda, David Flores-Peñaloza, and Sergio Rajsbaum. The topology of look-compute-move robot wait-free algorithms with hard termination. *Distributed Computing*, 32(3):235–255, 2019. doi:10.1007/s00446-018-0345-3.

4 Andreea B. Alexandru, Erica Blum, Jonathan Katz, and Julian Loss. State machine replication under changing network conditions. In *Advances in Cryptology – ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part I*, pages 681–710, Berlin, Heidelberg, 2023. Springer-Verlag. doi:10.1007/978-3-031-22963-3_23.

5 Dan Alistarh, Faith Ellen, and Joel Rybicki. Wait-free approximate agreement on graphs. In Tomasz Jurdziński and Stefan Schmid, editors, *Structural Information and Communication Complexity*, pages 87–105, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-79527-6_6.

6 Ananya Appan, Anirudh Chandramouli, and Ashish Choudhury. Perfectly-secure synchronous mpc with asynchronous fallback guarantees. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, pages 92–102, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519270.3538417.

7 Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 52–61, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/167088.167109.

8 Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 183–192, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/197917.198088.

9 Erica Blum, Jonathan Katz, and Julian Loss. Synchronous consensus with optimal asynchronous fallback guarantees. In *Theory of Cryptography*, volume 11891 of *Lecture Notes in Computer Science*, pages 131–150, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-36030-6_6.

10 Erica Blum, Jonathan Katz, and Julian Loss. Tardigrade: An atomic broadcast protocol for arbitrary network conditions. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part II*, volume 13091 of *LNCS*, pages 547–572. Springer, Heidelberg, December 2021. doi:10.1007/978-3-030-92075-3_19.

11 Erica Blum, Chen-Da Liu-Zhang, and Julian Loss. Always have a backup plan: Fully secure synchronous mpc with asynchronous fallback. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 707–731, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-56880-1_25.

12 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

13 Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 42–51, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/167088.167105.

14 Armando Castañeda, Sergio Rajsbaum, and Matthieu Roy. Convergence and covering on graphs for wait-free robots. *Journal of the Brazilian Computer Society*, 24(1):1, January 2018. doi:10.1186/s13173-017-0065-8.

15 Giovanni Deligios, Martin Hirt, and Chen-Da Liu-Zhang. Round-efficient byzantine agreement and multi-party computation with asynchronous fallback. In Kobbi Nissim and Brent Waters, editors, *Theory of Cryptography*, pages 623–653, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-90459-3_21.

16 Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, May 1986. doi:10.1145/5925.5931.

17 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983. doi:10.1137/0212045.

18 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

19 Matthias Fitzi and Juan A. Garay. Efficient player-optimal protocols for strong and differential consensus. In Elizabeth Borowsky and Sergio Rajsbaum, editors, *22nd ACM PODC*, pages 211–220. ACM, July 2003. doi:10.1145/872035.872066.

20 Diana Ghinea, Chen-Da Liu-Zhang, and Roger Wattenhofer. Optimal synchronous approximate agreement with asynchronous fallback. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, pages 70–80, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519270.3538442.

21 Diana Ghinea, Chen-Da Liu-Zhang, and Roger Wattenhofer. Multidimensional approximate agreement with asynchronous fallback. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '23, pages 141–151, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3558481.3591105.

22 Simon Holmgaard Kamp and Jesper Buus Nielsen. Byzantine agreement decomposed: Honest majority asynchronous atomic broadcast from reliable broadcast. Cryptology ePrint Archive, Paper 2023/1738, 2023. URL: https://eprint.iacr.org/2023/1738.

23 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982. doi:10.1145/357172.357176.

24 Jérémy Ledent. Brief announcement: Variants of approximate agreement on graphs and simplicial complexes. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 427–430, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467946.

25 Christoph Lenzen and Julian Loss. Optimal clock synchronization with signatures. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, pages 440–449, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519270.3538444.

26 Shihao Liu. The Impossibility of Approximate Agreement on a Larger Class of Graphs. In Eshcar Hillel, Roberto Palmieri, and Etienne Rivière, editors, *26th International Conference on Principles of Distributed Systems (OPODIS 2022)*, volume 253 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.OPODIS.2022.22.

27 Hammurabi Mendes and Maurice Herlihy. Multidimensional approximate agreement in byzantine asynchronous systems. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th ACM STOC*, pages 391–400. ACM Press, June 2013. doi:10.1145/2488608.2488657.

28 Hammurabi Mendes, Maurice Herlihy, Nitin Vaidya, and Vijay K Garg. Multidimensional agreement in byzantine systems. *Distributed Computing*, 28(6):423–441, 2015. doi:10.1007/S00446-014-0240-5.

29 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 31–42. ACM Press, October 2016. doi:10.1145/2976749.2978399.

30 Atsuki Momose and Ling Ren. Multi-threshold byzantine fault tolerance. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 1686–1699, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3460120.3484554.

31 Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with $t < n/3$ and $o(n^2)$ messages. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 2–9, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2611462.2611468.

32 Gil Neiger. Distributed consensus revisited. *Information Processing Letters*, 49(4):195–201, 1994. doi:10.1016/0020-0190(94)90011-6.

33 Thomas Nowak and Joel Rybicki. Byzantine Approximate Agreement on Graphs. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:17, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2019.29.

34 Thomas Nowak and Joel Rybicki. Byzantine approximate agreement on graphs, 2019. arXiv:1908.02743.

35 Gerard Sierksma. Caratheodory and helly-numbers of convex-product-structures. *Pacific Journal of Mathematics*, 61:275–282, 1975.

36 Lewis Tseng and Nitin H. Vaidya. Asynchronous convex hull consensus in the presence of crash faults. In Magnús M. Halldórsson and Shlomi Dolev, editors, *33rd ACM PODC*, pages 396–405. ACM, July 2014. doi:10.1145/2611462.2611470.

37 Nitin H. Vaidya and Vijay K. Garg. Byzantine vector consensus in complete graphs. In Panagiota Fatourou and Gadi Taubenfeld, editors, *32nd ACM PODC*, pages 65–73. ACM, July 2013. doi:10.1145/2484239.2484256.

Appendix

A Comparison with [33, Theorems 17 and 13]

In this section, we compare our impossibility results with the related [33, Theorems 17 and 13]. We find that our results generalize the aforementioned, with the exception of the first part of [33, Theorems 13], to which our findings are orthogonal. However, we exhibit an error in the proof of this part, rendering the result false in general.

To better understand these results, let us first introduce convex geometries. An abstract convexity space $\mathcal{C}$ on universe V is a *convex geometry* if it additionally satisfies that, for all convex sets $C \subsetneq V$, there exists $v \in V \setminus C$ such that $C \cup \{v\}$ is convex. This is a non-trivial requirement; e.g., $\mathbb{R}^D$ with straight-line convexity or box convexity is **not** a convex geometry. An example of a convex geometry is a chordal graph endowed with monophonic convexity. These notions are further discussed in the full version of our paper.

▶ **Theorem 21** ([33, Theorem 17]). *Let $\mathcal{C}$ be a convex geometry with Helly number ω. If the network is synchronous and $n \leq \omega \cdot t$, then no n-party protocol satisfies Convex Validity, Termination and Exact agreement.*

Contrasting this with Theorem 9, for the synchronous case our results generalize the previous by removing the strong requirement on $\mathcal{C}$ to be a convex geometry and by adding the fact that even agreement on at most $\min(n, \omega) - 1$ values is not possible.

Before continuing, we need to introduce a few more notions from convexity theory. One relevant notion will be that of *extreme* points. Namely, given a non-necessarily convex set $S \subseteq V$, the set $ex(S) = \{s \in S \mid \langle S \setminus s\rangle \subsetneq \langle S\rangle\}$ is the set of points in S any of whose removal would "shrink" the convex hull. Set S is called *free* if $\langle S\rangle = ex(S)$. Note that free sets are necessarily convex, as $\langle S\rangle = ex(S) \subseteq S \subseteq \langle S\rangle$, from which $\langle S\rangle = S$. Equivalently, S is free if and only if S is convex and $S = ex(S)$.

▶ **Theorem 22** ([33, Theorem 13]). *Let $\mathcal{C}$ be a convexity space with Helly number ω and Carathéodory number c. Assume the network is asynchronous and consider a protocol satisfying Convex Validity and termination, then:*

1. *If $n \leq (c+1) \cdot t$ there is an execution where the honest outputs do not form a free set in $\mathcal{C}$.*
2. *If $n \leq (\omega+1) \cdot t$ and $\mathcal{C}$ is a convex geometry, there is an execution where the set of honest outputs either has size at least ω or is not a free set in $\mathcal{C}$.*

Contrasting with Theorem 9, for the asynchronous case, our results generalize Part 2 of the above by once again removing the requirement on $\mathcal{C}$ to be a convex geometry and also by no longer requiring the clause "or is not a free set in $\mathcal{C}$." Our result also replaces ω by $\min(n, \omega)$, which we believe is also implicitly meant in the original result, as when $n < \omega$ the condition becomes vacuous, and a protocol where parties just output their own inputs satisfies Convex Validity and termination in some convex geometries.

Part 1 of Theorem 22, on the other hand, is orthogonal to our results. In our attempt to use adversarial families to potentially also recover Part 1, we have discovered an error in the proof of this part, making the result false in general. To display the error, we need to introduce one more notion: call a (not necessarily convex) subset $\mathcal{I} \subseteq V$ *irredundant* if there is a point $p \in \langle I\rangle$ such that the hull of no proper subset of $\mathcal{I}$ contains p. The Carathéodory number c of $\mathcal{C}$ is then the size of the largest such irredundant set $\mathcal{I}$. Now, the proof of Part 1 hinges on the following technical lemma:

▶ **Lemma 23** ([33, Lemma 15 of the full version [34]]). *Let $\mathcal{C}$ be a convexity space and A be an irredundant set such that $|A| > 1$. Then for any $a \in A$ and $y \in \langle A \rangle \setminus A$ there exists $b \in A \setminus \{a\}$ such that $y \notin \langle A \setminus \{b\} \rangle$.*

Note that we have added the condition "A is irredundant" missing from the original statement.[6] The error in the proof is towards the end where, using the original notation, it is stated that $y \notin \partial A = \langle A \rangle \setminus B \subseteq \langle A \rangle \setminus A$ implies that $y \notin \langle A \rangle \setminus A$, contradicting the hypothesis. However, in general, if some sets satisfy $S_1 \subseteq S_2$ and $y \notin S_1$ it does not follow that $y \notin S_2$. We next construct a convexity space where the lemma in fact fails for all irredundant sets A and all $a \in A$. First, introduce some auxiliary notation: given two convexity spaces $\mathcal{C}_1$ and $\mathcal{C}_2$ defined on universes V_1 and V_2 respectively, define $\mathcal{C}_1 \oplus \mathcal{C}_2$ to be the convexity space on universe $V_1 \times V_2$ such that $\mathcal{C}_1 \oplus \mathcal{C}_2 = \{C_1 \times C_2 \mid C_1 \in \mathcal{C}_1, C_2 \in \mathcal{C}_2\}$. For the construction, start with an arbitrary convexity space $\mathcal{C}$ on universe V and consider the convexity space $\mathcal{C}' = \mathcal{C} \oplus \{\varnothing, \{0,1\}\}$. To build intuition for $\mathcal{C}'$, notice that $\langle \{(v,i)\} \rangle = \{(v,0),(v,1)\}$ for any $v \in V$ and $i \in \{0,1\}$. Assume A is an irredundant set for $\mathcal{C}'$. Note that for no $v \in V$ does A contain both points $(v,0)$ and $(v,1)$, as otherwise it would be that $\langle A \rangle = \langle A \setminus \{(v,1)\} \rangle$, so A would not be irredundant. Consider any $a = (v,i) \in A$ and take $y = (v, 1-i) \in \langle A \rangle \setminus A$, then for any $b \in A \setminus \{a\}$ it holds that $a \in A \setminus \{b\}$, from which $\langle \{a\} \rangle = \{(v,0),(v,1)\} \subseteq \langle A \setminus \{b\} \rangle$, so $y \in \langle A \setminus \{b\} \rangle$, contradicting the statement of the lemma. Hence, we have constructed a space for which the lemma fails for any irredundant set A and any $a \in A$, indicating that any correct weakening of the lemma might sadly not be of much use in its current form.

We conclude by constructing a space whose Carathéodory number c is much larger than its Helly number ω, showing that our possibility results are not consistent with Part 2 of Theorem 22. To do so, we will use the fact [35, Theorems 2.1 and 3.2] that given convexity spaces $\mathcal{C}_1$ and $\mathcal{C}_2$ with Helly numbers ω_1, ω_2 and Carathéodory numbers c_1, c_2 the space $\mathcal{C}_1 \oplus \mathcal{C}_2$ has Helly number $\omega = \max\{\omega_1, \omega_2\}$ and Carathéodory number c satisfying $c_1 + c_2 - 2 \le c \le c_1 + c_2$. Consider the space $\mathcal{C} = \mathbb{R}^2$ with straight-line convexity, whose Helly and Carathéodory numbers are both 3. Then, the space $\mathcal{C}_k = \bigoplus_{\ell=1}^{k} \mathcal{C}$ has Helly number $\omega_k = 3$ and Carathéodory number $c_k \ge 3k - 2(k-1) = k+2$. For this space, our possibility results imply that, when the network is asynchronous, convex consensus be solved assuming $n > 4 \cdot t$, while Part 1 of Theorem 22 would imply that it can not be solved for $n \le (k+3) \cdot t$, which are incompatible statements for k large enough.

The keen-eyed reader might note that [33,34] require the universe to be finite, which is not the case for our counterexample. To also construct a counterexample with a finite universe, we will use the same technique, replacing $\mathbb{R}^2$ with straight-line convexity by a finite convexity space $\mathcal{X}$. The requirements for our technique to apply are mild since increasing k keeps the Helly number constant while strictly increasing the lower bound on the Carathéodory number, provided the Carathéodory number of $\mathcal{X}$ is at least 3. Hence, for k sufficiently large, the Carathéodory number of $\mathcal{C}_k = \bigoplus_{\ell=1}^{k} \mathcal{C} = \bigoplus_{\ell=1}^{k} \mathcal{X}$ will exceed its Helly number.[7] It remains to construct a finite $\mathcal{X}$ with Carathéodory number at least 3. This is not at all difficult: let the universe be four points $A, B, C, D \in \mathbb{R}^2$ such that A, B, C form an equilateral triangle and D is the center of the triangle, and the convexity notion be inherited from $\mathbb{R}^2$

[6] The proof of the lemma notes that if A is not irredundant the claim becomes vacuous, however, one can actually consider $\mathbb{R}^2$ with straight-line convexity, $A = \{(\pm 1, \pm 1)\}$ and $y = (0.5, 0.5)$, in which case $y \in \langle A \setminus \{a\} \rangle$ for any $a \in A$. This issue is however only minor since the lemma is only invoked in the proof of the subsequent [33, Lemma 16 of the full version [34]], where A is assumed to be irredundant.

[7] Note that for finite $\mathcal{X}$ the Helly number is always well-defined.

with straight-line convexity; i.e., $\langle\{A,B,C\}\rangle = \{A,B,C,D\}$. This space has a Carathéodory number of at least 3 because the set $\{A,B,C\}$ is irredundant (in fact exactly 3 because $\{A,B,C,D\}$ is not irredundant).

B Gather

In the following, we describe our protocol Π_{GTHR} realizing Theorem 19, restated below.

▶ **Theorem 19.** *If $n > 2 \cdot t_s + t_a$ and $t_a \leq t_s$, there is a (t_s, t_a)-secure GTHR protocol Π_{GTHR} (assuming PKI). The protocol has round complexity $O(1)$. If ℓ denotes the universe elements' size in bits and κ is the security parameter, it achieves a communication complexity of $O(n^3 \cdot \ell + n^4 \cdot \kappa)$ bits (can be reduced to $O(n^3 \cdot \ell + n^3 \cdot \kappa)$ with threshold signatures).*

As mentioned in Section 5, our protocol Π_{GTHR} adds one more step to the Overlap All-to-All Broadcast (oBC) protocol of [20], which we denote by Π_{oBC}. We need to highlight the difference between the definition of (t_s, t_a)-secure oBC presented in [20] and our definition of (t_s, t_a)-secure GTHR: oBC does not require t_s-Common Core. Instead, the oBC definition of [20] requires the weaker property t_s-Overlap:

***T*-Overlap**: If two honest parties P and P' terminate, then $\left|\mathcal{M}_P \cap \mathcal{M}_{P'}\right| \geq n - T$.

Until we reach the additional step, the protocols Π_{oBC} and Π_{GTHR} are identical. Π_{oBC} (and hence also Π_{GTHR}) heavily relies on the *witness technique* [1]. That is, in both protocols, parties send their values via Π_{rBC}. Then, they report to each other which values they received. When the values reported by a party P match the values received by a party P' via Π_{rBC}, P' marks P as a *witness*. In Π_{oBC}, parties are ready to terminate when (i) sufficient time has passed for honest values to be received via Π_{rBC} in a synchronous network, ensuring Honest Core; (ii) parties have gathered sufficient witnesses to ensure that every two honest parties P and P' have a common witness $P^\star$. This way, P and P' have received the same set of at least $n - t_s$ values reported by $P^\star$, and therefore the t_s-Overlap property holds. To achieve the superior guarantee t_s-Common Core required by GTHR in the asynchronous model, we will need a stronger termination condition as well.

Termination time in rBC. Before describing the protocol precisely, we need to include the rBC definition of [21], which makes the termination time explicit.

▶ **Definition 24** (Reliable Broadcast with explicit termination time)**.** *Let Π be a protocol where a designated party S (the sender) holds a value v_S, and every party P may output a value v_P. Consider the following properties:*

Validity*: If S is honest, and an honest party outputs v_P, then $v_P = v_S$.*

Consistency*: If P and P' are honest and output v_P and resp. $v_{P'}$, then $v_P = v_{P'}$.*

c-Honest Termination*: If S is honest, parties obtain outputs eventually. In addition, if the network is synchronous and the parties start executing the protocol at the same time τ, every honest party obtains output by time $\tau + c \cdot \Delta$.*

c′-Conditional Termination*: If an honest party P obtains output at time τ, then all honest parties obtain outputs eventually. In addition, if the network is synchronous and the honest parties start executing the protocol at the same time, then all honest parties obtain output by time $\tau + c' \cdot \Delta$.*

We say that Π is a (t_s, t_a, c, c')-secure Reliable Broadcast protocol if it achieves Validity, Consistency, c-Honest Termination, and c′-Conditional Termination even when t_s of the parties involved are corrupted if it runs in a synchronous network, and when up to t_a corruptions if it runs in an asynchronous network.

As mentioned in Section 5, we make use of the rBC protocol Π_{rBC} of Momose and Ren [30] described in the theorem below. The guarantees regarding explicit termination time follow from the analysis of [20].

▶ **Theorem 25** (Momose and Ren [30]). *Assume that $n > 2 \cdot t_s + t_a$ and $t_s \geq t_a$. Then, there is an n-party protocol achieving $(t_s, t_a, c_{\mathsf{rBC}}, c'_{\mathsf{rBC}})$-secure rBC (assuming PKI), where $c_{\mathsf{rBC}} := 3$ and $c'_{\mathsf{rBC}} := 1$. The protocol has round complexity $O(1)$. If ℓ denotes the universe elements' size in bits and κ is the security parameter, it achieves a communication complexity of $O(n^2 \cdot \ell + n^3 \cdot \kappa)$ bits. If, in addition, threshold signatures are available, the communication complexity reduces to $O(n^2 \cdot \ell + n^2 \cdot \kappa)$.*

Common steps of Π_{oBC} and Π_{GTHR}. We now describe the common steps of Π_{oBC} and Π_{GTHR} more precisely. Parties distribute their inputs via Π_{rBC}. When a party receives a value v from P via Π_{rBC}, it adds (v, P) to a set of value-party (or value-sender) pairs $\mathcal{M}$. Additionally, it adds P to a set W_0, representing *level-zero witnesses.* When at least $c_{\mathsf{rBC}} \cdot \Delta$ time has passed (meaning that, if the network is synchronous, every honest input was received), and when $|\mathcal{M}| \geq n - t_s$ (since at most t_s parties are corrupted), the parties reliably broadcast their set of level-zero witnesses W_0. Even after broadcasting W_0, parties may continue gathering level-zero witnesses. Then, if a party P receives a set of level-zero witnesses W'_0 from P' such that all values sent by parties in W'_0 were also received by P ($W'_0 \subseteq W_0$), P marks P' as a *level-one witness* by adding it to its set W_1.

Once each honest party gathers $n - t_s$ level-one witnesses, we have the guarantee that every pair of honest parties has a level-one witness in common. This means that every pair of honest parties has received $n - t_s$ common values via Π_{rBC}. This is the point where Π_{oBC} allows the parties to output the set of value-sender pairs obtained so far and terminate, as (t_s, t_a)-secure oBC is achieved.

Additional step in Π_{GTHR}. In contrast, to achieve the stronger property t_s-Common Core, our GTHR protocol continues: following the insights from the asynchronous GTHR protocol of [2], we obtain that, when $n - t_s$ honest parties hold sets W_1 of size $n - t_s$, then $t_s + 1$ honest parties have a common level-one *honest* witness $P^\star$. This will then enable us to achieve the t_s-Common Core property. Concretely, when party P gathers $n - t_s$ level-one witnesses, it sends its set W_1 to all the parties. P may continue marking parties as level-one witnesses even after sending its set W_1. When receiving a set W'_1 from some party P' such that $W'_1 \subseteq W_1$, P marks P' as a *level-two witness* by adding it to its set W_2. Once P collects $n - t_s$ level-two witnesses, it may output its set $\mathcal{M}$. This ensures that P has marked at least one of the parties that have reported sets W_1 containing $P^\star$ – and therefore P has marked $P^\star$ as a level-one witness as well. Hence, P has received the set $W^\star_0$ sent by $P^\star$, and therefore all the values sent by the parties in $W^\star_0$ have been included in P's set $\mathcal{M}$. This argument applies to every honest party, which ensures that the t_s-Common Core property holds.

We include the formal code of Π_{GTHR} below. Due to space constraints, we defer the analysis to the full version of our paper.

Protocol Π_{GTHR}

Code for party P with input v

```
$\tau_{\mathtt{start}} := \tau_{\mathtt{now}}$; $\mathcal{M} := \varnothing$; $W_0, W_1, W_2 := \varnothing$.
Send $v$ to every party via $\Pi_{\mathsf{rBC}}$.
Whenever receiving a value $v'$ from $P'$ via $\Pi_{\mathsf{rBC}}$, add $(v', P')$ to $\mathcal{M}$ and $P'$ to $W_0$.
If $\tau_{\mathtt{now}} \geq \tau_{\mathtt{start}} + c_{\mathsf{rBC}} \cdot \Delta$ and $|W_0| \geq n - t_s$:
    Send $W_0$ to all parties via $\Pi_{\mathsf{rBC}}$.
Whenever receiving $W_0'$ from $P'$ via $\Pi_{\mathsf{rBC}}$ such that $|W_0'| \geq n - t_s$:
    When $W_0' \subseteq W_0$, add $P'$ to $W_1$.
When $\tau_{\mathtt{now}} \geq \tau_{\mathtt{start}} + 2c_{\mathsf{rBC}} \cdot \Delta$ and $|W_1| \geq n - t_s$:
    Send $W_1$ to all parties.
Whenever receiving $W_1'$ from $P'$ such that $|W_1'| \geq n - t_s$:
    When $W_1' \subseteq W_1$, add $P'$ to $W_2$.
When $\tau_{\mathtt{now}} \geq \tau_{\mathtt{start}} + (2c_{\mathsf{rBC}} + c'_{\mathsf{rBC}}) \cdot \Delta$ and $|W_2| \geq n - t_s$:
    Output $\mathcal{M}$.
```

C Analysis of Π_{ACS}

We include the proof of Theorem 14, restated below.

▶ **Theorem 14.** *If $n > 2 \cdot t_s + t_a$ and $t_a \leq t_s$, there is a (t_s, t_a)-secure ACS protocol Π_{ACS} (assuming PKI). The protocol has expected round complexity $O(1)$. If ℓ denotes the universe elements' size in bits and κ is the security parameter, its expected communication complexity is $O(n^3 \cdot \ell + n^4 \cdot \kappa)$ bits. If threshold signatures are available, the expected communication complexity reduces to bits.*

Proof. In the following, we show that Π_{ACS} achieves (t_s, t_a)-secure ACS when $n > 2 \cdot t_s + t_a$.

First, the Validity and Consistency properties follow immediately from the Validity and Consistency properties of Π_{rBC} and Π_{GTHR}. Next, Π_{GTHR} ensures that all honest parties obtain sets $\mathcal{M}_{\mathsf{GTHR}}$ that intersect in at least $n - t_s$ values. In addition, if the network is synchronous, these sets contain all honest values, and are obtained simultaneously.

Therefore, if the network is synchronous, all parties join the Π_{BA} invocations simultaneously, hence all properties that Π_{BA} ensures when running in a synchronous network hold. It follows that all Π_{BA} invocations corresponding to honest parties result in output 1, and hence all honest values are included in the output sets $\mathcal{M}$, ensuring the Honest Core property. Moreover, parties agree on the same bit for every corrupted party. If the output bit for some corrupted party is 1, then at least one honest party has included this corrupted party's value in its set $\mathcal{M}_{\mathsf{GTHR}}$. By Observation 20, eventually, all honest parties receive this value as well. Therefore, all honest parties output the same set $\mathcal{M}$, hence Exact Agreement and Probabilistic Termination hold.

If the network is asynchronous, since Π_{GTHR} achieves Termination, all honest parties eventually join all Π_{BA} invocations and hence agree on a bit for each party. Π_{GTHR}'s t_s-Common Core property ensures that there exist at least $n - t_s$ invocations of Π_{BA} in which all honest parties input 1, and therefore output 1 in these invocations. For each invocation returning 1, the Weak Validity property ensures that at least one honest party P has input 1, meaning that P has received the corresponding value via Π_{GTHR}. Observation 20 then ensures that all parties eventually receive this value, and therefore all honest parties output the same set $\mathcal{M}$ of at least $n - t_s$ value-sender pairs, hence Exact Agreement, t_s-Output Size and Probabilistic Termination hold. ◀

A Simple Computability Theorem for Colorless Tasks in Submodels of the Iterated Immediate Snapshot

Yannis Coutouly
Aix Marseille University, France

Emmanuel Godard
Aix Marseille University, France

Abstract

The Iterated Immediate Snapshot model (IIS) is a central model in distributed computing. We present our work in the message adversary setting. We consider general message adversaries whose executions are arbitrary subsets of executions $\mathcal{M}$ of the IIS message adversary. We present a complete and explicit characterization of solvable colorless tasks given any submodel of IIS.

Based upon the geometrization mapping *geo* introduced in [8] to investigate set-agreement in general submodels, we give a simple necessary and sufficient condition for computability. The geometrization *geo* associates to any execution a point in $\mathbb{R}^N$. A colorless task $(\mathcal{I}, \mathcal{O}, \Delta)$ is solvable under $\mathcal{M}$ if and only if there is a continuous function $f : geo(skel^n(\mathcal{I}) \times \mathcal{M}) \longrightarrow |\mathcal{O}|$ carried by Δ.

This necessary and sufficient condition for colorless tasks was already known for full models like the Iterated Immediate Snapshot model [14, Th. 4.3.1] so our result is an extension of the characterization to any arbitrary submodels. It also shows the notion of continuity that is relevant for distributed computability of submodels is not the one from abstract simplicial complexes but the standard one from $\mathbb{R}^N$. As an example of its effectiveness, we can now derive the characterization of the computability of set-agreement on submodels from [8] by a direct application of the No-Retraction theorem of standard topology textbook. We also give a new fully geometric proof of the known characterization of computable colorless tasks for $t-$resilient layered snapshot model by using cross-sections of fiber bundles, a standard tool in algebraic topology.

2012 ACM Subject Classification Theory of computation → Computability; Theory of computation → Distributed algorithms

Keywords and phrases topological methods, geometric simplicial complex, set-agreement

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.16

1 Introduction

Distributed computability is the general investigation of which tasks could be solved in which distributed models. It is known since [12], [2] and [9] that some distributed tasks could have no algorithmic solution valid in all scenarios. Following the seminal works of Herlihy and Shavit [18], Borowsky and Gafni [5], Saks and Zaharoglou [28], using topological methods has proved very fruitful for distributed computing and for distributed computability in particular.

A distributed model that is widely used is the Iterated Immediate Snapshot (IIS) model, which is known to have the same task-computability power as the standard asynchronous read write wait-free model. In the setting of message adversaries, we consider general submodels $\mathcal{M}$ of the IIS model. These submodels correspond to arbitrary subsets of executions of IIS. We work on a subclass of distributed tasks, the *colorless tasks* : intuitively, any process can replace his input (resp. output) with the input (resp. output) of other processes while still correctly solving the task. Many important tasks like Consensus, $k-$set agreement, are colorless tasks. The ones needing to "break symmetry", like Election or Renaming, are not.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 16; pp. 16:1–16:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Related Work in Distributed Computability for IIS and Submodels

Distributed computability is a long time subject in distributed computing and topological methods are involved since introduced in [18, 28], see also the textbook [14] of Herlihy, Kozlov and Rajsbaum. We focus on this section on the previous results directly linked to our investigation, that is distributed computability for arbitrary submodels of the IIS model.

Models, Submodels and Message Adversaries. Topological methods have been first applied to the wait-free model, to be extended to some computability equivalent layered models, where these methods are more directly applicable. The Iterated Immediate Snapshot model appears now as a central model for distributed computing, either as a shared memory model or as a message adversary model.

We consider arbitrary subsets of executions of IIS, which is a setting that captures a wide range of models. Numerous submodels of the IIS model have already been investigated, they are usually called adversaries, see [14, Chap. 5] for results about colorless tasks for specific adversaries. Another category of adversaries are the affine adversaries of Kuznetsov and Rieutord [27, 22].

Our result is one of the few general results that can be applied to any submodel of the IIS model, which subsumes all models cited above. In this line of research, there have been works of Gafni, Kuznetsov and Manolescu in [10], and a recent extension by Attiya, Castañeda and Nowak in [3], following the work of Godard and Perdereau [11] for only two processes. In [3], a general computability theorem is presented for all submodels of the IIS model. Their results address both colorless and colored tasks, it is expressed using special infinite complexes called terminating subdivisions. This particular object capture non-uniform termination (to in part deal with non-compact sub-models) with infinite simplicial complexes. Since in this article we get rid of terminating subdivision we use the geometrization topology of [3].

Geometrization Topology. To include non-compact submodel we use the geometrization topology introduced by Godard and Perdereau in [11] for only two processes. It was later generalized by Coutouly and Godard in [8] to any number of processes in the case of general submodels of the Iterated Immediate Snapshot model. The geometrization mapping *geo* associates to any execution of IIS a point in $\mathbb{R}^N$. This induces a topology on IIS by considering as open sets the pre-images of open sets of $\mathbb{R}^N$. In [8], Coutouly and Godard only investigated the set-agreement task, not general colorless tasks. Moreover, the geometrization topology is mostly used in a descriptive way, not as a topology *per se*. This means that the geometrization topology is introduced in [8], but it is not actually used except as a way to provide some intuition to the combinatorial descriptions of some classes of executions. The main result of this paper, summarized in the following section, infers that there was actually more than an intuition, since we directly use this topology on sub model of IIS to express our computability results.

1.2 Our Contributions

A Generalized Computability Result. We build upon the *geo* mapping introduced by [8] and use it to express a new and simple colorless computability characterization. The *geo* mapping associates to any execution of IIS a point in $\mathbb{R}^N$ by considering an encoding by geometrical simplicial complexes. We adapt it to the colorless setting and a geometric universal colorless algorithm is presented, that is coined the Colorless Chromatic Average algorithm. The characterization is as follows in Theorem 11: a message adversary $\mathcal{M} \subset IIS_n$ solves a colorless task $(\mathcal{I}, \mathcal{O}, \Delta)$ if and only if there exists a continuous function $f : geo(skel^n(\mathcal{I}) \times \mathcal{M}) \longrightarrow |\mathcal{O}|$ carried by Δ.

This characterization was already known when $\mathcal{M} = IIS_n$, see [14, Thm. 4.3.1] and note that $geo(\mathcal{I} \times IIS_n) = |\mathcal{I}|$. Our result is therefore a wide extension of this simple topological characterization to any arbitrary submodel of the IIS model.

We discuss now the relevance of Theorem 11. The continuous function involved in this Theorem is continuous in the classical sense, i.e. for the topology of $\mathbb{R}^N$. In [3], a very general theorem is shown that relates computability to continuity of some function. However, this continuity is defined for a well chosen, but quite abstract and involved, topology on the set of executions. Here, thanks to the geometrization topology, we have basically to only deal with the standard continuity of the functions of $\mathbb{R}^N$, which actually appears more convenient, like in the Set-Agreement case, detailed below.

To compare to [3, Th 4.1] we focus less general task (only the colorless one) to remove the need from terminating subdivision. Both article have result on general adversaries of the IIS model. In this setting, a question of computability can be transformed to the existence of a continuous function between two classical topological spaces. For instance, we can directly use results from topology textbook as the No Retraction theorem [13, Cor. 2.15] to obtain a characterization for the Set-Agreement task. We also give a fully topological proof of the known characterization of the computable colorless tasks for the $t-$resilient layered snapshot protocol model by using cross-sections of fiber bundles.

These applications and their associated simple proofs, fully justifies, in our opinion, the move from abstract complexes to a fully geometric description of distributed systems by geometric simplicial complexes embedded in the ambient topology of $\mathbb{R}^N$. That is, the relevant simplicial complexes for distributed computability of colorless tasks are geometric simplicial complexes, seen as subspaces of $\mathbb{R}^N$ with its classical topology. It is known that abstract simplicial complex and geometric simplicial complex coincide when the complexes are of finite size. However, since dealing with general submodels, in particular so-called non-compact models, implies to associate a complex of possibly infinite size to distributed executions, we believe this is necessary.

Applications. As illustrations of our main computability theorem, we give new simple topological proofs of two known results : the characterization of submodels for which set-agreement is solvable (as already given in [8]) and the computability of colorless tasks against adversary. Our application are simple in the sense that we only use textbook theorems for "classical" topological spaces (like the standard Euclidian space $\mathbb{R}^N$ or the standard ball $\mathbb{S}^n$).

We investigate colorless tasks for so-called *adversary models*. These are sub-models of IIS where the failures can be not homogeneous: there is an arbitrary list F of sets of processes that can fail simultaneously, F is assumed to be inclusion closed.

A common example of adversary submodels is the $t-$faulty submodel which is a setting where at most t processes will eventually crash. In our IIS setting, this corresponds to the $t-$resilient layered snapshot protocol model. This is a well studied model, since [17] it is known such model cannot solve the $t-$set agreement. Nowadays, we have a nice topological understanding thanks to [15], a task is solvable in a $t-$resilient layered snapshot model if and only there is a continuous map from $|skel^t(\mathcal{I})| \to |\mathcal{O}|$ carried by Δ. From this, it can be deduced that t-resilient model cannot solve the k-set-Agreement task unless $k > t$. This result was also obtained in [4] using an algorithmic construction. A nice overview can be found in [21], and a comprehensive investigation in [14, Chap. 5]. We present here a new topological proof of these results that exploits a new topological and geometric interpretation of the reduction between models by using fiber bundles and cross sections, which are standard notion of topological spaces.

2 Models of Computation and Definitions

2.1 Models of Computation

We introduce our notations. Let $n \in \mathbb{N}$, we consider systems with $n+1$ processes. We denote $\Pi_n = [0,..,n]$ the set of processes. Since sending a message is an asymmetric operation, we will work with directed graphs. We use standard directed graph (or digraph) notations : given G, $V(G)$ is the set of vertices, $A(G) \subset V(G) \times V(G)$ is the set of arcs.

▶ **Definition 1.** *We denote by $\mathcal{G}_n$ the set of directed graphs with vertices in Π_n.*

A dynamic graph $\mathbf{G}$ *is a sequence $G_1, G_2, \cdots, G_r, \cdots$ where G_r is a directed graph with vertices in Π_n. We also denote by $\mathbf{G}(r)$ the digraph G_r. A* message adversary *is a set of dynamic graphs. Since that n will be mostly fixed through the paper, we use Π for the set of processes and $\mathcal{G}$ for the set of graphs with vertices Π when there is no ambiguity.*

Intuitively, the graph at position r of the sequence describes whether there will be, or not, transmission of some messages sent at round r. A formal definition of an execution under a dynamic graph will be given in Section 2.3. We will use the standard following notations in order to describe more easily our message adversaries [25].

A dynamic graph is seen as a infinite word over the alphabet $\mathcal{G}$.Given $U \subset \mathcal{G}$, U^* is the set of all finite sequences of elements of U, U^ω is the set of all infinite ones and $U^\infty = U^* \cup U^\omega$.

Given $\mathbf{G} \in \mathcal{G}^\omega$, if $\mathbf{G} = \mathbf{HK}$, with $\mathbf{H} \in \mathcal{G}^*, \mathbf{K} \in \mathcal{G}^\omega$, we say that $\mathbf{H}$ is *a prefix* of $\mathbf{G}$, and $\mathbf{K}$ *a suffix*. $Pref(\mathbf{G})$ denotes the set of prefixes of $\mathbf{G}$. A message adversary of the form U^ω, with $U \subset \mathcal{G}$, is called an *oblivious adversary* or an *iterated adversary*. A word in $\mathcal{M} \subset \mathcal{G}^\omega$ is called a *communication scenario* (or *scenario* for short) of message adversary $\mathcal{M}$. Given a word $\mathbf{H} \in \mathcal{G}^*$, it is called a *partial scenario* and $len(\mathbf{H})$ is the length of this word. The prefix of $\mathbf{G}$ of length r is denoted $\mathbf{G}_{|r}$ (not to be confused with $\mathbf{G}(r)$ which is the r-th letter of $\mathbf{G}$, it is the digraph at time r).

We show how standard fault environments are conveniently described in our framework.Consider a synchronous system of two processes $\circ$ and $\bullet$ where at most one of the processes can crash, the associated adversary is the following (using rational expression): $C_1 = \{\circ\!\leftrightarrow\!\bullet^\omega\} \cup \{\circ\!\leftrightarrow\!\bullet\}^*(\{\circ\!\leftarrow\!\bullet^\omega, \circ\!\rightarrow\!\bullet^\omega\})$. In the system of two processes $\circ$ and $\bullet$ where, at each round, only one message can be lost, the associated message adversary is $\{\circ\!\leftrightarrow\!\bullet, \circ\!\leftarrow\!\bullet, \circ\!\rightarrow\!\bullet\}^\omega$.

2.2 Iterated Immediate Snapshot Message Adversary

The previous example is IIS_1, we now detail the main message adversary we consider. Given a graph G, we denote by $In_G(a) = \{b \in V(G) \mid (b,a) \in A(G)\}$ the set of incoming vertices of a in $V(G)$. A graph G has the *containment Property* if for all $a, b \in V(G)$, $In_G(a) \subset In_G(b)$ or $In_G(b) \subset In_G(a)$. We say that a graph G has the *Immediacy Property* if for all $a, b, c \in V(G)$, $(a,b), (b,c) \in A(G)$ implies that $(a,c) \in A(G)$.

▶ **Definition 2** ([14])**.** *We set $ImS_n = \{G \in \mathcal{G}_n \mid G$ has the Immediacy and Containment properties $\}$. The Iterated Immediate Snapshot message adversary for $n+1$ processes is the message adversary $IIS_n = ImS_n^\omega$.*

The Iterated Immediate Snapshot model was first introduced as a (shared) memory model and then has been shown to be equivalent to the above message adversary first as tournaments and iterated tournaments [6, 1], then as this message adversary [14, 16]. See also [26] for a survey of the reductions involved in these layered models.

2.3 Execution of a Distributed Algorithm

Given a message adversary $\mathcal{M}$ and a set of initial configurations $\mathcal{I}$, we define what is an execution of a given algorithm $\mathcal{A}$ subject to $\mathcal{M}$ with initialization $\mathcal{I}$. An execution is constituted of an initialization step, and a (possibly infinite) sequence of rounds of messages exchanges and corresponding local state updates. When the initialization is clear from the context, we will use *scenario* and *execution* interchangeably.

An execution of an algorithm $\mathcal{A}$ under scenario $w \in \mathcal{M}$ and initialization $\iota \in \mathcal{I}$ is the following. This execution is denoted $\iota.w$. First, ι affects the initial state to all processes of Π. Then the system progresses in rounds. A round is decomposed in 3 steps : sending, receiving, updating the local state. At round $r \in \mathbb{N}$, messages are sent by the processes using the `SendAll()` primitive. The fact that the corresponding receive actions, using the `Receive()` primitive, will be successful depends on $G = w(r)$, G is called the *instant graph* at round r.

Let $p, q \in \Pi$. The message sent by p is received by q on the condition that the arc $(p, q) \in A(G)$. Then, all processes update their state according to the received values and $\mathcal{A}$. Note that it is assumed that p always receives its own value, that is $(p, p) \in A(G)$ for all p and G. However, in examples, this might be implicit for clarity and brevity.

Let $w \in \mathcal{M}, \iota \in \mathcal{I}$. Given $u \in Pref(w)$, we denote by $\mathbf{s}_p(\iota.u)$ the state of process p at the $len(u)$-th round of the algorithm $\mathcal{A}$ under scenario w with initialization ι. This means that $\mathbf{s}_p(\iota.\varepsilon) = \iota(p)$ represents the initial state of p in ι, where ε denotes the empty word.

3 Task Definition

We start by restating some standard definitions of combinatorial topology.

▶ **Definition 3** (Abstract simplicial complex). *[14, Def 3.2.1] Let V be a set, and C a collection of finite subsets of V. C is an abstract simplicial complex on V if*

1. $\forall \sigma \in C, \forall \tau \subseteq \sigma$, *we have* $\tau \in C$;
2. $\forall v \in V, \{v\} \in C$.

An element of V is a *vertex* of C and $V(C)$ denotes the set of vertices of C. A set $\sigma \in C$ is a *simplex* where *dim* σ is the number of vertices in σ minus one. We say that σ is a *facet* if there is no other simplex that contains σ. If $C_1 \subseteq C_2$ then we say that C_1 is a *subcomplex* of C_2, a complex is *pure* if all facets have the same dimension. The pair (C_1, χ_{C_1}) is a *chromatic complex* if C_1 is a complex and the function $\chi_{C_1} : V(C_1) \to \Pi$ has the property that $\forall \sigma \in C_1, \forall v_1, v_2 \in V(\sigma), v_1 \neq v_2 \Leftrightarrow \chi_{C_1}(v_1) \neq \chi_{C_1}(v_2)$.

The *border* of a simplex σ, is $\partial(\sigma) = \{\tau \in \sigma | dim(\tau) = \dim(\sigma) - 1\}$. A *$\ell$-skeleton* of C_1 is the collection of the simplices of dimension equal or less than ℓ, we write $skel^\ell(C_1)$. The *star* of a simplex $\sigma \in C_1$ is $St(\sigma, C_1) = \bigcup_{\tau \in C_1, \sigma \subseteq \tau} \tau$, the *extended star* is $St^*(\sigma, C_1) = \bigcup_{v \in \sigma} St(v, C_1)$.

▶ **Definition 4** (Simplical map). *[14, Def 3.2.2] Let C_1, C_2 be two simplicial complexes, a simplicial map is a map $\Phi : V(C_1) \to V(C_2)$ such that $\forall \sigma \in C_1, \Phi(\sigma) \in C_2$.*

▶ **Definition 5** (Carrier map). *[14, Def 3.4.1] Let C_1, C_2 be two simplicial complexes, a carrier map $\Phi : C_1 \to 2^{C_2}$ is a mapping such that $\forall \sigma, \tau \in C_1$, and $\sigma \subseteq \tau$ imply $\Phi(\sigma) \subseteq \Phi(\tau)$.*

In addition, a carrier map $\Phi : C_1 \to 2^{C_2}$ is *rigid* when $\forall \sigma \in C_1, dim(\sigma) = d, \Phi(\sigma)$ is a pure complex of dimension d. A simplicial map $\varphi : C_1 \to C_2$ is *carried* by Φ if $\forall \sigma \in C_1, \varphi(\sigma) \in \Phi(\sigma)$. A carrier map is *chromatic* if it is rigid and $\forall \sigma \in C_1, \chi_{C_1}(\sigma) = \chi_{C_2}(\varphi(\sigma))$ where $\chi_{C_2}(\varphi(\sigma)) = \{\chi_{C_2}(v) | v \in V(\varphi(\sigma))\}$. We say that V_{in} is the domain of input values and V_{out} the domain of output values.

▶ **Definition 6** (Colorless Task). *[14, Def 4.2.1] A* colorless task *is a triple* $(\mathcal{I}, \mathcal{O}, \Delta)$ *where :*

- $\mathcal{I}$ *is the input complex, where each simplex is a subset of* V_{in}*,*
- $\mathcal{O}$ *is the output complex, where each simplex is a subset of* V_{out}*,*
- $\Delta : \mathcal{I} \to 2^{\mathcal{O}}$ *is a carrier map that encodes the specification of the task.*

In [14, Chap 4.1], the notion of colorless protocol is presented both operationally and combinatorially. We will give the corresponding geometric version in Algorithm 1.

▶ **Definition 7.** *An algorithm* $\mathcal{A}$ *solves a colorless task* $(\mathcal{I}, \mathcal{O}, \Delta)$ *for the message adversary* $\mathcal{M}$ *if for any* $\iota \in \mathcal{I}$*, any scenario* $w \in \mathcal{M}$*, there exist* u *a prefix of* w *such that the state of the system* $\{\mathbf{s}_0(\iota.u), \ldots, \mathbf{s}_n(\iota.u)\} = out$ *satisfies the specification of the task, ie* $out \in \Delta(\iota)$*.*

4 Geometric Definition of Simplicial Complexes

4.1 Standard Definitions

In this paper, we actually handle simplicial complexes as geometric complexes, so we present the standard definitions of simplicial complexes in the geometric setting [24]. We fix $N \in \mathbb{N}$. We note $\mathbf{B}(x, r) = \{y \in X | d(x, y) \leq r\}$ with $x \in \mathbb{R}^N, r \in \mathbb{R}$ and $d(x, y)$ the Euclidean distance on $\mathbb{R}^N$.

▶ **Definition 8** (Geometric Simplex). *Let* $n \in \mathbb{N}$*. A finite set* $\sigma = \{x_0, \ldots, x_n\} \subset \mathbb{R}^N$ *is called a* simplex *of dimension* n *if the vectors* $\{x_1 - x_0, \ldots, x_n - x_0\}$ *are linearly independent.*

We denote by $|\sigma|$ the convex hull of σ and $Int(\sigma)$ is the interior of $|\sigma|$. We denote $\mathbb{S}^n$ “the” simplex of dimension n : through this paper we assume a fixed embedding in $\mathbb{R}^N$ for $\mathbb{S}^n = (x_0^*, \ldots, x_n^*)$. We will also assume that its diameter $diam(\mathbb{S}^n)$ is 1. We usually associate χ such that $\chi(x_i^*) = i$, to get the chromatic simplex $\mathbb{S}^n$.

▶ **Definition 9** ([24]). *A* simplicial complex *is a collection* C *of* simplices *such that :*

(a) *If* $\sigma \in C$ *and* $\sigma' \subset \sigma$*, then* $\sigma' \in C$*,*

(b) *If* $\sigma, \tau \in C$ *and* $|\sigma| \cap |\tau| \neq \emptyset$ *then there exists* $\sigma' \in C$ *such that*

- $|\sigma| \cap |\tau| = |\sigma'|$*,*
- $\sigma' \subset \sigma, \sigma' \subset \tau$*.*

We denote $\wr C \wr = \bigcup_{S \in C} |S|$, this is the *geometrization of* C. Note that the geometrization here should not be confused with the standard *geometric realization*. They are the same at the set level but not at the topological level. A discussion in Appendix A providesx more information on this subject. Since the difference only appears for infinite complexes, we will still denote $|\sigma|$ the convex hull of a simplex σ, instead of $\wr\sigma\wr$.

We use the same terminology as for abstract complexes, with some additionals concepts. Let A and B be simplicial complexes. A map $f \colon V(A) \to V(B)$ defines a *simplicial map* if it preserves the simplices, *i.e.* for each simplex σ of A, the image $f(\sigma)$ is a simplex of B. By linear combination of the barycentric coordinates, f extends to the *barycentric map* $\wr f \wr \colon \wr A \wr \to \wr B \wr$. This can be done by taking any simplex $\sigma = \{x_0, \ldots, x_n\}$ of A. Since any $y \in |\sigma|$ is obtained as $y = \sum_{i=0}^n t_i.x_i$ with $t_i \in [0, 1]$ and $\sum_{i=0}^n t_i = 1$, we set $\wr f \wr(y) = \sum_{i=0}^n t_i.f(x_i)$.

For any geometric chromatic simplex $\sigma = (v_0, v_1, \ldots, v_n)$ (ie with a fixed order on the set of vertices), we have an unique affine map called the *characteristic map* $\varphi_\sigma : |\mathbb{S}^n| \to |\sigma|$ taking the i^{th} vertex of $\mathbb{S}^n$ to v_i. This is indeed the barycentric map of the simplicial map taking the i^{th} vertex of $\mathbb{S}^n$ to v_i.

Let $X \subset \mathbb{R}^N$, a function $f : X \to |C_2|$ respects a carrier map $\Delta : C_1 \to 2^{C_2}$ with $X \subseteq \wr C_1 \wr$, if $\forall \sigma \in C_1, f(|\sigma| \cap X) \subseteq \Delta(\sigma)$. The *open star* of $\sigma \in C_1$: $St^\circ(\sigma, C_1) = \bigcup_{\tau \in C_1, \sigma \subseteq \tau} Int(\tau)$.

▶ **Definition 10** (Subdivision). *[14, Def 3.6.1] Let C_1, C_2 be two geometric simplicial complexes. We say that C_2 is a subdivision of C_1 if : $\wr C_1 \wr = \wr C_2 \wr$, and each simplex of C_1 is the union of finitely many simplices of C_2.*

4.2 Geometric Encoding of Iterated Immediate Snapshots Configurations

Here we present the mapping *geo* that links points of $\mathbb{R}^N$ and executions of the Iterated Immediate Snapshot model. Since this has been introduced in [8], this is only sketched here. The reader can refer to Appendix B for all the technical details in the setting of this paper.

There are two equivalent ways to define *geo*. It can be seen as the limit value of running a specific algorithm, called *the Chromatic Average algorithm*. Or, for a given execution w, it can be seen as the limit of iterating the Standard Chromatic Subdivision along the simplices corresponding to the successive instant graphs $w(r)$. The only difference with [8], is that we have to adapt to the setting of colorless algorithms by introducing the *Reduced Chromatic Average algorithm*. But the ideas and proof techniques are in essence the same as [8].

Algorithm 1 The reduced version of Chromatic Average Algorithm for process i with initial value $x_i^* \in \mathbb{R}^N$.

```
$x \leftarrow x_i^*$;
Loop forever
    SendAll($x$);
    $V$ ←Receive() // set of all received values including its own;
    $d \leftarrow sizeof(V) - 1$ // the process received $d$ values, excluding its own ;
    $x = \frac{1}{2d+1}x + \sum_{y \in V \setminus \{x\}} \frac{2}{2d+1}y$;
EndLoop
```

We consider Algorithm 1, which is an adaptation of the Chromatic Average algorithm of [8]. As proved in [8] and in Appendix B, it is possible to show that the values x of all processes converge to the same limit $geo(w)$ for any execution $w \in IIS_n$. It is related to the known fact that the standard chromatic subdivision is *mesh-shrinking* [14].

4.3 A Topology for IIS_n

We present the *geometrization topology* on the set of execution of IIS as introduced in [8]. It is the topology induced by geo^{-1} from the standard topology in $\mathbb{R}^N$.

The *geometrization topology* is defined on IIS_n by considering as open sets the sets $geo^{-1}(\Omega)$ where Ω is an open set of $\mathbb{R}^N$. A collection of sets can define a topology when any union of sets of the collection is in the collection, and when any finite intersection of sets of the collection is in the collection. This is straightforward for a collection of inverse images of a collection that satisfies these properties. Note this also makes *geo* continuous by definition.

By considering the definition of *geo* from the iterations of the Standard Chromatic subdivision, we also have $geo(IIS_n) = |\mathbb{S}^n|$. Now, we want to associate a geometric point to any execution $w \in IIS_n$ with a specific initial configuration ι. Hence, we extend the construction on the simplex $\mathbb{S}^n$ to any simplicial complex $\mathcal{I}$ in the following way: $\forall \iota \in \mathcal{I}, \forall w \in \mathcal{M}$, $geo(\iota, w) = \varphi_\iota(geo(w))$, where φ_ι is the characteristic map of ι, mapping $\mathbb{S}^{dim(\iota)}$ to ι. We define $geo(\mathcal{I} \times \mathcal{M}) = \bigcup_{w \in \mathcal{M}, \sigma \in \mathcal{I}} \varphi_\sigma(geo(w))$. This construction into the set of execution allow us to associate to any message adversary $\mathcal{M} \subseteq IIS$ a topological space in $\mathbb{R}^N$.

5 A Generalisation of the Asynchronous Computability Theorem

The main result of this paper is an extension to any submodel of IIS of the result [14, Thm. 4.3.1] about computability of colorless tasks in IIS. Our proof follows the same line as [14] with adaptation of some key tools. We first express the main result.

▶ **Theorem 11** (Colorless-GACT). *Let $(\mathcal{I}, \mathcal{O}, \Delta)$ be a colorless task. This is solvable on $\mathcal{M} \subseteq IIS_n$ if and only if there is a continuous function $f : geo(skel^n\mathcal{I} \times \mathcal{M}) \to |\mathcal{O}|$ carried by Δ.*

The rest of this section is a long proof of the main result. We prove this equivalence in four inductive steps starting from the right hand side of the above theorem. We only give here the outline of the proof, that is the properties we want to prove equivalent. All proper definitions stated here will be introduced along the way :

1. A continuous function $f : geo(\mathcal{I} \times \mathcal{M}) \to |\mathcal{O}|$ satisfies an η-star condition for some function η.
2. From this η-star condition, we construct a IIS-terminating subdivision and a semi-simplicial approximation of f
3. This semi-simplicial approximation of f yield an algorithm solving the task $(I, \mathcal{O}, \Delta)$
4. An algorithmic solution for $\mathcal{M}$ implies the existence of a continuous map $geo(\mathcal{I} \times \mathcal{M}) \to |\mathcal{O}|$

For the rest of this section, n is fixed, we note $\mathcal{I}$ instead of $skel^n\mathcal{I}$. We also fix $X \subseteq \mathbb{R}^N$. We will set $X = geo(\mathcal{I} \times \mathcal{M})$ in the end. Let $\mathcal{O}$ be a finite simplicial complex.

5.1 From continuous function to η-star condition

We adapt the notion of *star-condition.*

▶ **Definition 12** (Star Condition for η). *Let $\eta : X \longrightarrow]0, +\infty[$ and let $f : X \to |\mathcal{O}|$, f satisfies the star condition for η if $\forall x \in X, \exists v \in V(\mathcal{O}), f(\mathbf{B}(x, \eta(x)) \cap X) \subseteq St^\circ(v)$.*

We also say f satisfies the $\eta-$star condition when we have a given η for the star condition above. See Figure 3 in Appendix D for an illustration.

▶ **Proposition 13.** *Let $f : X \to |\mathcal{O}|$ a continuous function. Then there is $\eta : X \longrightarrow]0, +\infty[$ such that f satisfies the η-star condition.*

Proof. We recall the standard definition of continuity : $\forall x \in X, \forall \epsilon > 0, \exists \delta_\epsilon(x) > 0$ such that $\forall x_0, x_0 \in \mathbf{B}(x, \delta_\epsilon(x)) \Rightarrow f(x_0) \in \mathbf{B}(f(x), \epsilon)$. Let $y \in |\mathcal{O}|$ and $\sigma_y \in \mathcal{O}$ the simplex of minimal dimension such that $y \in |\sigma_y|$. Let $\epsilon(y) = d(y, |\mathcal{O}| \setminus St^\circ(\sigma_y))$. We know that $\epsilon(y) \neq 0$ because $y \in St^\circ(\sigma_y)$, which is an open space. From there, the η-star condition is obtained with $\eta(x) = \delta_{\epsilon(f(x))}(x)$ since $f(\mathbf{B}(x, \eta(x)) \cap X) \subseteq \mathbf{B}(f(x), \epsilon(f(x))) \subseteq St^\circ(\sigma_{f(x)})$. ◀

5.2 From η-star condition to semi-simplicial approximation

We say that a simplicial complex is compatible with a subspace X if it covers X entirely with every simplex needed for such a cover.

▶ **Definition 14** (Complex compatible with a subspace). *Let $X \subseteq \mathbb{R}^N$ and C a simplicial complex. We say that C is compatible with X if $X \subseteq \wr C \wr$, and for all facet σ of C, $|\sigma| \cap X \neq \emptyset$.*

We use the notion of terminating subdivisions, that were introduced in [10, 3]. Here we present a more complete and explicit definition, this is needed since we are in the geometric context. The intuition behind this construction is that we want to associate a complex with the set of executions of a given algorithm on $\mathcal{M}$. As the termination of this algorithm could be not uniform (even with a fixed initial configuration), the associated complex may be of infinite size. Since a subdivision of a simplex cannot be infinite, we have to define an adapted construction from the iterated application of the Standard Chromatic subdivision.

Given a complex C, let $C(T) = \bigcup_{\sigma \in C, V(\sigma) \subseteq T} \sigma$ with $T \subseteq V(C)$ to represent the subcomplex of C formed by the vertices in T. Moreover, $JOIN(C_1, C_2) = \{|\sigma \cup \tau| | \sigma \in C_1, \tau \in C_2\}$ is the usual join of simplices [19]. We define $EChr$ as the following operator, given C and $T \subseteq V(C)$. Intuitively, the vertex marked as terminated are in T. We note $U = V(C) \setminus T$. The operator $EChr$ subdivides with the standard chromatic subdivision the facets that are fully in U, does not modify the ones that are fully in T and subdivides in an adequate way the facets in between.

$$EChr(T, C) = (\bigcup_{\sigma \in C} Chr\,\sigma(U)) \cup (\bigcup_{\sigma \in C} JOIN(Chr\,\sigma(U), \sigma(T))) \qquad (1)$$

▶ **Definition 15** (IIS-Terminating subdivision)**.** *Let $\mathcal{I}$ a simplicial complex. The sequences $C_0, C_1, \ldots$ (collection of simplices) and $T_0, T_1, \ldots$ (collection of increasing set of vertices) form a IIS-*terminating subdivision of $\mathcal{I}$, *if we have for all $i \in \mathbb{N}$:*

1. $C_0 = \mathcal{I}, T_0 = \emptyset$
2. $C_{i+1} \subseteq EChr(T_i, C_i)$
3. $T_i \subseteq V(C_i)$

We say that $\bigcup C_i(T_i)$ is an IIS-terminating subdivision complex. This is actually a simplicial complex, as proved in Appendix C.

A simplicial approximation is a standard topological construct that is the basis of the proof technique of the similar computability theorem in [14, Th 4.3.1]. A simplicial approximation is a simplicial function that approximates (in some sense) a function where the domain and the co-domain are simplicial complexes. We have to adapt this definition since here we only have that the co-domain as a simplicial complex.

▶ **Definition 16** (semi-simplicial approximation)**.** *Let $f : X \to |\mathcal{O}|$ a function. The function $\psi : V(C) \to V(\mathcal{O})$ is a semi-simplicial approximation for f if C a IIS-terminating subdivision compatible with X, and ψ is a simplicial map such that $\forall \sigma \in C, f(St^\circ(\sigma) \cap X) \subseteq St^\circ(\psi(\sigma))$.*

In order to construct an IIS-terminating subdivision of $\mathcal{I}$ compatible with X, we need a predicate to “set in a terminating state”. Let $\eta : X \longrightarrow]0, +\infty[$, C a simplicial complex and $v \in V(C)$, we define $\mathbf{P}_\eta(v, C) = \{\exists x \in X, | \ |St(v, C)| \subseteq \mathbf{B}(x, \eta(x))\}$. Let $C_0 = \mathcal{I}$ the simplicial complex to subdivide, $U_0 = V(C_0)$, $T_0 = \emptyset$. For all $i \in \mathbb{N}$ we set :

1. $D_{i+1} = EChr(T_i, C_i)$
2. $C_{i+1} = \{\sigma \in D_{i+1} | \ |\sigma| \cap X \neq \emptyset$ and σ is a facet of $D_{i+1}\}$
3. $T_{i+1} = \{v \in V(C_{i+1}) | \mathbf{P}_\eta(v, C_{i+1})\}$

The final complex is $C_\eta = \bigcup_{i \in \mathbb{N}} C_i(T_i)$. In Figure 1 the vertices marked in red are the ones in T_i. On the left, the subdivided simplex is the one without vertices in red. On the right, some simplices are added with a $JOIN$ operation. Then the simplices in blue and in dotted lines are the ones that will be removed at step 2, since they do not intersect X.

▶ **Proposition 17.** *C_η is a simplicial complex.*

Proof. We have D_i is a simplicial complex. In C_i, removing facets of D_i still yields a simplicial complex. ◀

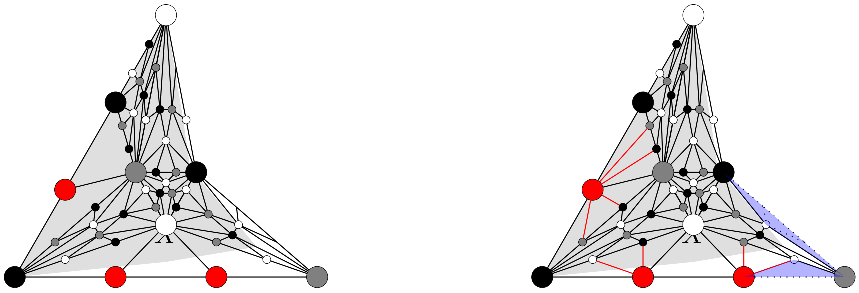


Figure 1 Construction of the IIS-Terminating Subdivision compatible with a space X.

▶ **Proposition 18.** *The subdivision C_η is compatible with X.*

Proof. For the inclusion property, $\forall x \in X$, since $x \in |C_0|, \forall r \in \mathbb{N}, \exists \sigma_r \in C_r$ such that $x \in |\sigma_r|$. The subdivision operator Chr is mesh-shrinking, this means that $\exists r_0 \in \mathbb{N}, \forall v \in V(\sigma_{r_0}), mesh(St(v, C_{r_0})) < \eta(x)$. Then $St(v, C_{r_0}) \subseteq B(x, \eta(x))$, which means that all vertex of σ_{r_0} are in T_{r_0} hence σ_{r_0} is in C_η so $x \in \wr C_\eta \wr$ and $X \subseteq \wr C_\eta \wr$. Since we only remove facet $\sigma \in C_\eta$ such that $|\sigma| \cap X = \emptyset$ in the second step of the construction we have the compatibility of C_η with X ◀

We can now construct a semi-simplicial approximation with C_η.

▶ **Proposition 19.** *Let $\eta : X \longrightarrow]0, +\infty[$ and let $f : X \to |\mathcal{O}|$ a function that satisfies the η-star condition, then f has a semi-simplicial approximation $\psi_\eta : V(C_\eta) \to V(\mathcal{O})$.*

Proof. Let C_η be the IIS-terminating subdivision of $\mathcal{I}$ defined above from η. Let σ a simplex of C_η, v a vertex of $V(\sigma)$, Since $\mathbf{P}_\eta(v, C_\eta)$ is satisfied, $\exists x_v \in X$ such that $|St(v, C_\eta)| \subseteq \mathbf{B}(x_v, \eta(x))$. By the η-star property we have that $\exists y_v \in V(\mathcal{O}), f(\mathbf{B}(x_v, \eta(x_v)) \cap X) \subseteq St^\circ(y_v, \mathcal{O})$. Let $\psi_\eta(v) = y_v$. Let's prove that ψ_η is indeed a semi-simplicial approximation.

We know that $\forall \sigma \in C_\eta, \forall v \in V(\sigma), \mathbf{P}_\eta(v, C_\eta)$ is true, then we have that : $\bigcap_{v \in V(\sigma)} |St(v, C_\eta)| \subseteq \bigcap_{v \in V(\sigma)} \mathbf{B}(x_v, \eta(x_v))$. The η-star condition gives that $\bigcap_{v \in V(\sigma)} f(\mathbf{B}(x_v, \eta(x_v))) \subseteq St^\circ(y_v, \mathcal{O})$. By noticing that $St^\circ(v, C_\eta) \subseteq |St(v, C_\eta)|$, we can combine theses inclusions and obtain that : $\bigcap_{v \in V(\sigma)} f(St^\circ(v, C_\eta)) \subseteq \bigcap_{v \in V(\sigma)} St^\circ(y_v, \mathcal{O})$ since $\bigcap_{v \in V(\sigma)} St(v) = St(\sigma)$.

This can be rewritten as : $f(St^\circ(\sigma, C_\eta)) \subseteq \bigcap_{v \in V(\sigma)} St^\circ(\psi_\eta(v), \mathcal{O})$, which is the property of the definition 16. Furthermore, because C_η is compatible with X we have that $\exists x \in X, x \in |\sigma|$. Since $|\sigma| \subseteq |St(\sigma)|$ which implies $f(x) \in |\psi_\eta(\sigma)|$ then $\bigcap_{v \in V(\sigma)} St^\circ(\psi_\eta(v), \mathcal{O})$ is non-empty therefore $\psi_\eta(\sigma)$ is a simplex, the function ψ_η is simplicial. ◀

We need to prove that the semi-simplicial approximation $\psi_\eta : V(C_\eta) \to V(O)$ of f is carried by the carrier map of f.

▶ **Lemma 20** (semi-simplicial approximation and carrier map)**.** *Let $\eta : X \longrightarrow]0, +\infty[$ and let $f : X \to |\mathcal{O}|$ a continuous function that respects $\Delta : \mathcal{I} \to 2^{\mathcal{O}}$ a carrier map. Then the semi-simplicial approximation $\psi_\eta : C_\eta \to \mathcal{O}$ of f respects also Δ.*

Proof. Let $\sigma \in C_\eta$, with $\sigma = \{v_0, v_1 \dots v_k\}$, $\psi_\eta(\sigma) = \{y_0, y_1, \dots y_k\}$ and $\{x_0, x_1, \dots x_k\}$ a points of X such that $|St(v_i)| \subseteq \mathbf{B}(x_i, \eta(x_i))$. Also, we have that $f(|\sigma|) \subseteq |\Delta(\sigma)|$ because f respects Δ and by construction of $C_\eta, f(\mathbf{B}(x_i, \eta(x_i))) \subseteq St^\circ(y_i)$. By way of contradiction, assume that $\psi_\eta(v_i) \notin \Delta(\sigma)$, by the $\mathbf{P}_\eta$ property we have that x_i covers $|St(v_i)|$. When we apply the function f, we obtain that $f(|\sigma|) \subseteq f(\mathbf{B}(x_i, \eta(x_i)) \cap X) \subseteq St^\circ(\psi_\eta(v_i))$. We can remark that $\psi_\eta(v_i) \notin \Delta(\sigma) \Rightarrow St^\circ(\psi_\eta(v_i)) \nsubseteq \Delta(\sigma)$. We can conclude that $f(\sigma) \nsubseteq \Delta(\sigma)$, which contradicts our hypothesis. ◀

5.3 From semi-simplicial approximation to an algorithm

Now we show that a semi-simplicial approximation can be used to define an algorithm.

▶ **Proposition 21.** *Let $f : geo(\mathcal{I} \times \mathcal{M}) \to |\mathcal{O}|$ a continuous function which respects a carrier Δ then the task $(\mathcal{I}, \mathcal{O}, \Delta)$ is solvable by an algorithm in $\mathcal{M}$.*

We define the following Algorithm 2 from the Chromatic Averaging Algorithm and using C_η and ψ_η, x_p is the initial position of p in the complex $\mathcal{I}$.

Algorithm 2 $\mathcal{A}_{\psi_\eta}$: algorithm derived from $\psi_\eta : C_\eta \to \mathcal{O}$.

```
$x \leftarrow x_p$;
$i \leftarrow 0$;
while $x \notin T_i$ do
    $i \leftarrow i + 1$;
    SendAll($x$);
    $V \leftarrow$Receive() // set of all received values including its own;
    $d \leftarrow sizeof(V) - 1$ // the process received $d$ values, excluding its own ;
    $x = \frac{1}{2d+1}x + \sum_{y \in V \setminus \{x\}} \frac{2}{2d+1}y$;
end
return $\psi_\eta(x)$;
```

▶ **Proposition 22.** *The Algorithm $\mathcal{A}_{\psi_\eta}$ terminates for all executions in $\mathcal{I} \times \mathcal{M}$.*

Proof. We set $X = geo(\mathcal{I} \times \mathcal{M})$ and the corresponding C_η is compatible with X which implies that $X \subseteq \wr C_\eta \wr$. Then $\forall w \in \mathcal{I} \times \mathcal{M},\ geo(w) \in X$. We can deduce that $\exists \sigma \in C_\eta,\ geo(w) \in |\sigma|$. By construction of C_η, $\exists i \in \mathbb{N}$ such that $x \in T_i$, hence every processes terminate. ◀

▶ **Proposition 23.** *Algorithm $\mathcal{A}_{\psi_\eta}$ respects the specification described by Δ*

Proof. The decision is given by ψ_η, a semi-simplicial approximation of f that respects Δ. ◀

5.4 From an algorithm to a continuous function

We conclude this proof by constructing a continuous function from a given algorithm. We will need to normalize this algorithm first. In the colorless setting, a normalized algorithm is an algorithm where when a process sees any decision value, it decides instantly one of these values. If an algorithm is correct, its normalized version is also correct for this colorless task.

▶ **Proposition 24.** *Let $\mathcal{A}$ a normalized algorithm for solving the task $(\mathcal{I}, \mathcal{O}, \Delta)$ in the submodel $\mathcal{M}$ then there exists a continuous function $f : geo(skel^n(\mathcal{I}) \times \mathcal{M}) \to |\mathcal{O}|$ that respects Δ.*

Proof. An algorithm solving a task $(\mathcal{I}, \mathcal{O}, \Delta)$ provides a decision function $\varphi_\mathcal{A} : Y \to V(\mathcal{O})$, where $Y \subset |skel^n(\mathcal{I}|$ (Y is the set of vertices of $Chr^r(\mathcal{I})$, for all r). We use this decision function $\varphi_\mathcal{A}$ to construct a IIS-terminating subdivision. Let $C_0, C_1, C_2, \ldots$ be a sequence of complexes and $T_0, T_1, T_2, \ldots$ a sequence of vertices of these complexes. We fix $T_0 = \emptyset$ and $C_0 = \mathcal{I}$ which immediately satisfies the condition 1). A vertex is added in T_i if the algorithm decide on the couple $(Process, View)$ at the round i. Since every decision of process are permanent we have the properties 3) and 4) of IIS-terminating subdivision. At the round i of the algorithm we construct the complex C_i using the $EChr$ operator, this operation corresponds to one round of IIS while allowing non-uniform termination and compatibility with $\mathcal{M}$. If we take $C_{\mathcal{A}} = \bigcup_{i\in\mathbb{N}} C_i(T_i)$, this correspond to the set of processes that will terminate in our algorithm and yields a IIS-terminating subdivision. Furthermore, $C_{\mathcal{A}}$ is compatible with X, because $\varphi_{\mathcal{A}}$ is a decision function on every execution of $\mathcal{I} \times \mathcal{M}$ then we have that $X \subseteq C_{\mathcal{A}}$. Also, if $\sigma \in C_{\mathcal{A}}, |\sigma| \cap X = \emptyset$ then this means that we decide on a execution that is not in $\mathcal{M}$, which is outside of our algorithm scope. Hence $C_{\mathcal{A}}$ is compatible with X. Since every vertex in $V(C_{\mathcal{A}})$ correspond to an execution in $\mathcal{I} \times \mathcal{M}$, from $\varphi_{\mathcal{A}}$ we obtain a simplicial function $\varphi : V(C_{\mathcal{A}}) \to V(\mathcal{O})$ that respect Δ.

Then we have that $\varphi(St(v, C_{\mathcal{A}})) \subseteq St(\varphi(v), \mathcal{O})$. We can now classically extend the simplicial function φ to a function $\varphi_C : \wr C_{\mathcal{A}} \wr \to |\mathcal{O}|$ by linear extension on the barycentric coordinates. This extension guaranties that φ_C respects Δ : $\varphi_C(|\sigma|) \subseteq |\Delta(\sigma)|$ since $\varphi(\sigma) \in \Delta(\sigma)$. We need to prove the continuity[1] of $\varphi_C : \forall x \in \wr C_{\mathcal{A}} \wr, \exists \sigma \in C_{\mathcal{A}}, x \in |\sigma|, \exists r \in \mathbb{N}, V(\sigma) \subseteq T_r$ and $V(\sigma) \nsubseteq T_{r-1}$. Then if $x \in Int(\sigma)$ the continuity can be obtained directly because the barycentric extension is always continuous on a given simplex. If $x \in \partial(\sigma)$ then because the algorithm is normalized we have that $\forall v \in V(\sigma), St(v, C_{r+1}(T_{r+1})) = St(v, C_{r+1})$. Since this a finite simplicial complex we obtain the continuity with a barycentric extension. ◀

6 Application to Set-Agreement

We give here a direct, and therefore simpler than [8], proof for general set-agreement computability. For all $n \in \mathbb{N}$, the *set-agreement problem* is defined by the following properties [23]. Given initial *init* values in $[0, n]$, each process outputs a value such that

Agreement the size of the set of output values is at most n,
Validity the output values are initial values of some processes,
Termination All processes terminates.

▶ **Theorem 25** ([8]). *It is possible to solve Set-Agreement on $\mathcal{M} \subset IIS_n$ iff $geo(\mathcal{M}) \neq |\mathbb{S}^n|$.*

Proof. We denote by $(\mathcal{I}_{sa}, \mathcal{O}_{sa}, \Delta_{sa})$ the colorless task for set-agreement. We have $skel^n \mathcal{I}_{sa} = \mathbb{S}^n$. For the necessary condition, we first get from Thm. 11 that there exists a continuous function from $geo(\mathbb{S}^n \times \mathcal{M})$, *i.e.* a continuous function from $geo(\mathcal{M})$ to $|\mathcal{O}_{sa}|$. $\mathcal{O}_{sa}$ is $\partial\mathbb{S}^n$, the boundary of $\mathbb{S}^n$. The No Retraction theorem [13, Cor. 2.15] states that there is no continuous function from $|S^n|$ to $|\partial S^n|$. This means that $geo(\mathcal{M})$ cannot be equal to $|\mathbb{S}^n|$.

In the reverse direction, if $geo(\mathcal{M}) \subsetneq |\mathbb{S}^n|$, we note x_0 a point in $|\S^n|$ that is not in $geo(\mathcal{M})$. We can construct in a standard way a continuous function from $geo(\mathcal{M})$ to $|\partial S^n|$ by using x_0 as a base point to "project" points x of $geo(\mathcal{M})$ onto $|\partial\mathbb{S}^n|$: the image of x is the intersection of the half-line $x_0 x$ with $|\partial\mathbb{S}^n|$ (in the special case where $x_0 \in |\partial\mathbb{S}^n|$, we project on the complex obtained by removing from ∂S^n the simplexes that contains x_0) . ◀

This characterization is quite expected, it is known the No-Retraction theorem is the topological obstruction for solving set-agreement in models such as the Iterated Immediate Snapshot model. We underline that this proof is way simpler that the proof of Coutouly and Godard in [8], that used Sperner and König lemmas in very involved ways. We underline that having at least one missing point from $|\mathbb{S}^n|$, ie a hole in $geo(\mathcal{I}_{sa} \times \mathcal{M})$, does not mean that $\mathcal{M}$ is $\mathcal{G}_n^\omega$ minus one execution. Since geo is not injective, many executions could be removed, that is all executions that maps to some x_0. These pre-images are called $geo-$classes, they are fully described in [8]. Some $geo-$classes are of infinite size when $n \geq 2$.

[1] we emphasize that the underlying topology here is not that of standard geometric realizations of complexes, therefore being simplicial does not imply the linear extension to be continuous in the sense we have to prove here.

7 Application to Adversaries Submodels

An adversary, in the sense of [14, Chap. 5.4], is a message adversary where the executions are exactly defined by the set of possible simultaneous failures.

Formally, we say that a process p is influencing a time t a process q, if there is a sequence of messages starting at time t from p that eventually reaches q. Finally, given $w \in IIS_n$, we denote by $Q(w)$ the set of processes that are influencing infinitely many times in w all the other processes. In IIS_n, this set is always non empty. In the message adversary setting, the set of "failed" processes is the set $\Pi \setminus Q(w)$. An adversary A is defined by a set $F(A)$ of subsets of Π that is inclusion-closed. The set of corresponding executions is denoted as $\mathcal{M}_A = \{w \in IIS_n | \exists P \in F(A), Q(w) = \Pi \setminus P\}$. A well investigated case is the adversary R_t, given $t \leq n$, where $F(R_t)$ is the set of subsets of size at most t. It is the $t-$resilient layered immediate snapshot protocol submodel.

As in [14, Chap. 5.4], we define a core as a minimal set of processes that will not all fail in any execution. For the $t-$resilient layered snapshot protocol model, a core is any subset of size $t + 1$. Even if processes are independent of the set of input values in the colorless setting, we will be able to assign a set of input value to any core C. Hence, we choose a core $C = \{p_0, \dots p_c\}$ of size $c + 1$ and we will construct an application $\pi_c^* : \mathcal{I} \times \mathcal{M}_A \to |skel^c\mathcal{I}|$.

Let G a graph of ImS_n, given a set of vertices C, we denote by $G[C]$ the subgraph induced by C, that is $V(G[C]) = C$ and $E(G[C]) = E(G) \cap (C \times C)$. We extend this notation to executions, $\forall w \in IIS_n,$, with $w = G_1, G_2, \dots,$ we set $w[C] = G_1[C], G_2[C], \dots$.

The function π_c^* is constructed by applying this reduction for a chosen C_w to every execution of $\mathcal{M}_A$. We fix an order on the processes. We set C_w to be the set $\Pi \setminus Q(w)$ together with the q lowest processes of $Q(w)$, where $q = c + 1 - |\Pi \setminus Q(w)|$. The set C_w is always of size $c + 1$ and is therefore not in $F(A)$. Finally, we set $\pi^*(w) = geo(w[C_w])$.

▶ **Proposition 26.** *The function $\pi_c^* : \mathcal{I} \times \mathcal{M}_A \to |skel^c\mathcal{I}|$ has the following properties :*

1. *it is continuous and surjective,*
2. $\forall w, w' \in \mathcal{M}_A$, $geo(w) = geo(w') \Rightarrow \pi_c^*(w) = \pi_c^*(w')$

Proof. The property (1) is directly obtained by construction. For the second property, first we remark that $geo(w) = geo(w')$ implies $Q(w) = Q(w')$. Indeed, consider $p \in \Pi$, such that $p \in Q(w)$. The process p cannot distinguish w from w' otherwise all other processes will eventually distinguish the executions. It means that the set of processes that influence p infinitely many times is the same in both executions. By definition, this set includes $Q(w')$ in the execution w'. Since p is influencing infinitely many times all processes in w, and influence is transitive, we have $Q(w') \subset Q(w)$. Symmetrically, we get $Q(w) \subset Q(w')$.

Therefore $C_w = C_{w'}$. We conclude by a simple case by case analysis from the different cases where $geo(w) = geo(w')$ as given in [8, Th. 25]. ◀

So we can also define a function $\pi : geo(\mathcal{I} \times \mathcal{M}_A) \to |skel^c\mathcal{I}|$ by setting $\pi(x) = \pi_c^*(w)$, where w is any element of $geo^{-1}(x)$. We will now show that a restriction of π actually enjoys a very interesting topological property. First, we give a standard definition.

▶ **Definition 27** (Fiber Bundle [13]). *Let E, B, F topological spaces. (E, B, π, F) is a fiber bundle with base B and fiber F if $\pi : E \to B$ is a continuous surjection such that for every $x \in B$, there is an open neighborhood $U \subseteq B$ of x such that there is a homeomorphism $\varphi : \pi^{-1}(U) \to U \times F$, and $U \times F$ is the product space in such a way that π agrees with the projection $proj$ onto the first factor,* i.e. $\pi_{|\pi^{-1}(U)} = \varphi \circ proj$.

One example of a classic example of fiber bundle is in Appendix E. We say that a run $w \in \mathcal{M}_A$ is *special* if w has a suffix w' (after step j) where all the instant graphs have their sources in $Q(w)$ and for all $i \geq j$ $G(i)$ is such that the arcs between $Q(w) \cap C_w$ and its complement $Q(w) \setminus C_w$ are all from $Q(w) \cap C_w$ when i is even, and to $Q(w) \cap C_w$ when i is odd. We denote Spe_A the set of special runs of $\mathcal{M}_A$.

▶ **Proposition 28.** *The function $\pi : E \longrightarrow skel^c(\mathcal{I})$ is a fiber bundle with $E = geo(Spe_A)$, $B = skel^c(\mathcal{I})$ and $F = \mathbb{S}^{n-c-1}$.*

Proof. For any point $x \in |skel^c\mathcal{I}|$, there is a special run w in $\mathcal{M}_A$ such that $geo(w) = x$. Indeed, given C a core of size $c+1$ with $c+1$ different initial values, it is possible to complement the execution $w^* \in IIS_C$ such that $\pi(geo(w^*)) = x$ in a special way : after the step j^* where only processes in $Q(w^*)$ influence all others in C, in instant graph G_i, $i \geq j^*$, processes from $\Pi \setminus C$ have an arc to processes in $C \setminus Q(w^*)$ and arcs between $\Pi \setminus C$ and $Q(w^*)$ alternate direction if i is even or odd. The arcs between processes of $\Pi \setminus C$ can be any pattern from $ImS_{\Pi \setminus C}$. This means that the restriction of π on E is surjective.

Now we focus on the neighborhood condition. As previously, we consider $x \in |skel^c\mathcal{I}|$, and the corresponding w^* and j^*. We set U to be the neighbourhood of x where executions share the prefix of w^* up to step j^*. In the previous section, we have said that we can define w by complementing w^* choosing any pattern in $ImS_{\Pi \setminus C}$. We remark that there is actually no other way to complement w^* to get a special execution. So the fiber $\pi^{-1}(x)$ is homeomorphic to $geo(IIS_{\Pi \setminus C})$, that is exactly $\mathbb{S}^{n-c-1}$. ◀

Concluding, from the main theorem, a colorless task $(\mathcal{I}, \mathcal{O}, \Delta)$ is solvable on $\mathcal{M}_A$ if and only if there exists a continuous function $f : geo(\mathcal{I} \times \mathcal{M}_A) \longrightarrow |\mathcal{O}|$ carried by Δ. We will show that this is equivalent to the existence of a continuous function $g : |skel^c\mathcal{I}| \longrightarrow |\mathcal{O}|$ carried by Δ so we can get an alternative and fully topological proof of the following.

▶ **Theorem 29** ([14, Th.5.4.3]). *A colorless task $(\mathcal{I}, \mathcal{O}, \Delta)$ is solvable on $\mathcal{M}_A$ for an adversary A with a core of size c if and only if there exists a continuous function $g : |skel^c\mathcal{I}| \longrightarrow |\mathcal{O}|$ carried by Δ.*

Proof. We show that the existence of f is equivalent to the existence of g. We assume $c < n$ otherwise the statement are equal and g is f. We start with the easy direction, assuming there exists g a continuous function $g : |skel^c\mathcal{I}| \longrightarrow |\mathcal{O}|$ carried by Δ. For a given facet S of $skel^n\mathcal{I}$, since $c < n$, there exists $x_1 \in |S|$ such that $x_1 \notin geo(S \times \mathcal{M}_A)$, it is therefore possible to have a retract from $|S| \setminus \{x_1\}$ onto $|Skel^{n-1}(S)|$. We can repeat this until reaching $|Skel^c(S)|$. We consider μ the composition of this sequence of retracts of $|Skel^n(\mathcal{I})| \setminus \{x_1, x_2, ...\}$ onto $|skel^cS|$. We set $f = g \circ \mu$. Such f is continuous by composition. Since this is a retract, μ is the identity on $|skel^cS|$ and f is carried by Δ.

Now, we assume that we have a continuous function $f : geo(\mathcal{I} \times \mathcal{M}_A) \longrightarrow |\mathcal{O}|$ carried by Δ. We would like to define $g = f \circ s$ where s would be a kind of right inverse for π as defined above. In order to show that, we will use the fact that π is a fiber bundle for E and $B = |skel^c\mathcal{I}|$. In the context of fiber bundles, what we are looking for is called a (cross) section s, that is, a continuous function $s : B \longrightarrow E$ such that $\pi \circ s = Id_B$. Cross-sections do not always exist, however since the fiber F is $\mathbb{S}^d$, we get that there is indeed a section s, see e.g. [7, Cor. 7.13], as a corollary of Whitehead Obstruction theorem. Since f is continuous, g is also continuous. By construction of π, g is also carried by Δ on $skel^c\mathcal{I}$ since f also is. ◀

We have this immediate corollary for the $t-$resilient layered snapshot protocol model R_t.

▶ **Corollary 30.** *Let $t \leq n$. A colorless task $(\mathcal{I}, \mathcal{O}, \Delta)$ is solvable on R_t if and only if there exists a continuous function $g : |skel^t\mathcal{I}| \longrightarrow |\mathcal{O}|$ carried by Δ.*

8 Conclusion

In this work, we have presented a simple characterization of computability of colorless tasks for any submodels of the IIS model. We believe that this theorem will have many applications, from simpler proof of known results to new characterisation of some colorless tasks. Note also that it is possible to extend the presented technique to submodels of models corresponding to mesh-shrinking subdivisions (like the barycentric subdivision), we underline it would change the definition of *geo*, therefore this would not mean that a colorless task would be solvable for the same submodels. Together with the kind of classical topology approaches that we have shown to be effective in the two applications suggest that this work opens many perpespective to investigate computability in more general distributed models.

Since we are actually using the geometrization topology in this paper, we complement the remarks from [8] by some important points about this topology. In a topological space, a neighbourhood for point x is an open set containing x. The set of neighbourhoods of x is denoted N_x. A topological space is said to satisfy the T_0 separation axiom if $x \neq y \Longrightarrow N_x \neq N_y$. When $N_x = N_y$, we say that x and y are not (topology) distinguishable.

Since the topology we are building upon for $\wr IIS_n \wr$ is the one induced by the standard space $\mathbb{R}^N$, which satisfies T_0, via the geo^{-1} mapping, it is straightforward to see that non-distinguishable sets are exactly the geo-equivalence classes that are not singletons, since any neighbourhood of w in the geometrization topology will be a neighbourhood of w', when $geo(w) = geo(w')$. A description of theses geo-equivalence classes can be found in [8], and it is shown that there always exists non-singleton classes. By construction, the topology on IIS_n is therefore not T_0. However, if we quotient this space by the classes of indistinguishability, which is called the Kolmogorov quotient, we obtain a topological space homeomorphic to $|\mathbb{S}^n|$. So up to Kolmogorov quotient, the topology introduced here for investigating colorless tasks on IIS_n can be considered *classical*.

We are also looking forward to address colored tasks by an extension of these results. Since it is known that a statement like Thm. 11 is not strong enough for some non-coloured task, there needs to have some additional conditions in the theorem statement. Another line of research would be to characterize, in a topological way, the colored tasks that admit a characterization à la Thm. 11, related to a better understanding of the relationship between the set of executions seen as a topological space with the geometrization topology, which is quite simple, and seen as a topological space with the general topology of [3].

References

1 Yehuda Afek and Eli Gafni. *Asynchrony from Synchrony*, pages 225–239. Number 7730 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013.

2 Dana Angluin. Local and global properties in networks of processors (extended abstract). In Raymond E. Miller, Seymour Ginsburg, Walter A. Burkhard, and Richard J. Lipton, editors, *Proceedings of the 12th Annual ACM Symposium on Theory of Computing, April 28-30, 1980, Los Angeles, California, USA*, pages 82–93. ACM, 1980. doi:10.1145/800141.804655.

3 Hagit Attiya, Armando Castañeda, and Thomas Nowak. Topological characterization of task solvability in general models of computation. In Rotem Oshman, editor, *Proceedings of the 37th International Symposium on Distributed Computing (DISC'23)*, volume 281 of *LIPICS*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. To appear. doi:10.4230/LIPICS.DISC.2023.5.

4 E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001. doi:10.1007/PL00008933.

5 Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 91–100, New York, NY, USA, 1993. ACM Press. doi:10.1145/167088.167119.

6 Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computation (extended abstract). In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '97, pages 189–198. ACM, 1997. doi:10.1145/259380.259439.

7 Ralph L Cohen. Bundles, homotopy, and manifolds. *Lecture notes – Standford University*, 2023.

8 Yannis Coutouly and Emmanuel Godard. A topology by geometrization for sub-iterated immediate snapshot message adversaries and applications to set-agreement. In Rotem Oshman, editor, *Proceedings of the 37th International Symposium on Distributed Computing (DISC'23)*, volume 281 of *LIPICS*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. To appear. doi:10.4230/LIPICS.DISC.2023.15.

9 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

10 Eli Gafni, Petr Kuznetsov, and Ciprian Manolescu. A generalized asynchronous computability theorem. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 222–231. ACM, 2014. doi:10.1145/2611462.2611477.

11 Emmanuel Godard and Eloi Perdereau. Back to the coordinated attack problem. *Math. Struct. Comput. Sci.*, 30(10):1089–1113, 2020. doi:10.1017/S0960129521000037.

12 Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag. doi:10.1007/3-540-08755-9_9.

13 Allen Hatcher. *Algebraic Topology*. Cambridge University Press, 2002.

14 Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 2013.

15 Maurice Herlihy and Sergio Rajsbaum. The topology of shared-memory adversaries. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 105–113. Association for Computing Machinery, 2010. doi:10.1145/1835698.1835724.

16 Maurice Herlihy, Sergio Rajsbaum, and Michel Raynal. Computability in distributed computing: A tutorial. *SIGACT News*, 43(3):88–110, 2012. doi:10.1145/2421096.2421118.

17 Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 111–120. ACM, 1993. doi:10.1145/167088.167125.

18 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999. doi:10.1145/331524.331529.

19 Dmitry N. Kozlov. *Combinatorial Algebraic Topology*, volume 21 of *Algorithms and computation in mathematics*. Springer, 2008. doi:10.1007/978-3-540-71962-5.

20 Dmitry N. Kozlov. Chromatic subdivision of a simplicial complex. *Homology Homotopy Appl.*, 14(2):197–209, 2012. URL: http://projecteuclid.org/euclid.hha/1355321488.

21 Petr Kuznetsov. Understanding non-uniform failure models. *Bull. EATCS*, 106:53–77, 2012. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/80.

22 Petr Kuznetsov, Thibault Rieutord, and Yuan He. An asynchronous computability theorem for fair adversaries. In Calvin Newport and Idit Keidar, editors, *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 387–396. ACM, 2018. URL: https://dl.acm.org/citation.cfm?id=3212765.

23 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

24 James R. Munkres. *Elements Of Algebraic Topology*. Addison Wesley Publishing Company, 1984.

25 J.E. Pin and D. Perrin. *Infinite Words*, volume 141 of *Pure and Applied Mathematics*. Elsevier, 2004.

26 Sergio Rajsbaum. Iterated shared memory models. In Alejandro López-Ortiz, editor, *LATIN 2010: Theoretical Informatics*, pages 407–416, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-12200-2_36.

27 Thibault Rieutord. *Combinatorial characterization of asynchronous distributed computability. (Caractérisation combinatoire de la calculabilité distribuée asynchrone)*. PhD thesis, University of Paris-Saclay, France, 2018. URL: https://tel.archives-ouvertes.fr/tel-02938080.

28 M. Saks and F. Zaharoglou. "wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. on Computing*, 29:1449–1483, 2000. doi:10.1137/S0097539796307698.

A A Counter-Example about Geometric Realizations

We remind the reader that the *geometrization* of C, denoted $\wr C \wr$, that is the union of the convex hulls $|\sigma|$ of the simplices σ of C, is endowed with the standard topology from $\mathbb{R}^N$.

This should not be confused with the geometric realization, that is endowed of what is called a weak topology.

In this section, we provide an example of a simplicial complex whodse topology as a geometric realization is different from the topology it has as geometrization, that is in the ambient R^N space. That means that there exists infinite complex for which the topological spaces $\wr C \wr$ and $|C|$ are not necessarily homeomorphic. This is actually quite well known, see e.g. [19]. This example can actually be translated exactly to the distributed executions that exhibit an error from [10] in [11, Sect. 5.1].

The example is given with $N = 1$ but that can be generalized to any N. We consider $C = \{0\} \cup \{[\frac{1}{r+1}, \frac{1}{r}] \mid r \in \mathbb{N}^*\}$.

We denote $|C|$ the topological space of C defined as a geometric realization. The closed sets of $|C|$ are the sets F such that $F \cap S$ is closed (in $\mathbb{R}$) for all $S \in C$, see [24]. Therefore $|C|$ has two connected components. We have $F =]0, 1]$ is closed in $|C|$ since $F \cap [\frac{1}{r+1}, \frac{1}{r}] = [\frac{1}{r+1}, \frac{1}{r}]$, hence is closed for all r. Moreover, $F \cap \{0\} = \emptyset$ which is also closed in $\mathbb{R}$. We also have that $\{0\}$ is closed in $|C|$, so C can be covered by two disjoint closed sets, it is therefore not connected.

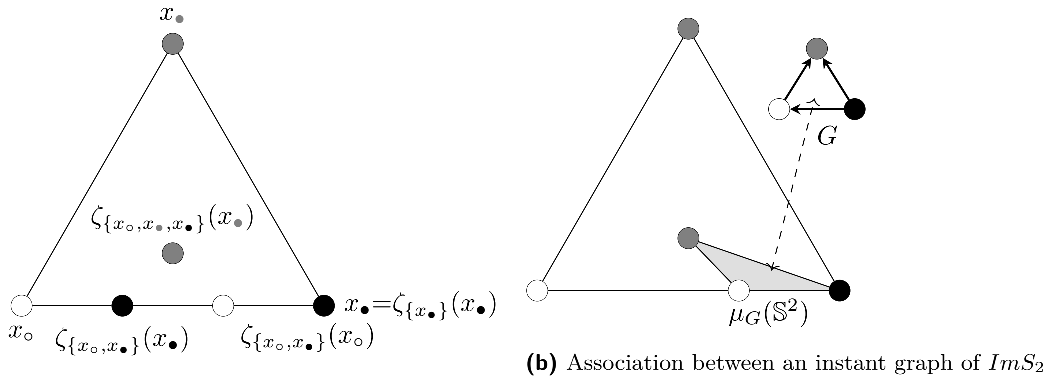
On the other end, at the set level, $\wr C \wr$ is exactly $[0, 1]$. So within the standard ambient topology of $\mathbb{R}$, $\wr C \wr$ is connected.

Since they do not have the same number of connected components, the two spaces $\wr C \wr$ and $|C|$ cannot be homeomorphic.

This type of problem can happend in many distributed situation, as in [11] :let $\mathcal{M}_1 = IIS \setminus \{\{\circ\!\leftrightarrow\!\bullet, \circ\!\leftarrow\!\bullet^\omega\}, \{\circ\!\rightarrow\!\bullet, \circ\!\leftarrow\!\bullet^\omega\}\}$ and $\mathcal{M}_2 = IIS \setminus \{\circ\!\rightarrow\!\bullet, \circ\!\leftarrow\!\bullet^\omega\}$ to remark that only $\mathcal{M}_1$ can solve the binary consensus task.

B The Standard Chromatic Subdivision

Here we present the standard chromatic subdivision, [14] and [19], as a geometric complex. We start with chromatic subdivisions.


$x_{\textcolor{gray}{\bullet}}$
$\zeta_{\{x_\circ, x_{\textcolor{gray}{\bullet}}, x_\bullet\}}(x_{\textcolor{gray}{\bullet}})$
$x_\bullet = \zeta_{\{x_\bullet\}}(x_\bullet)$
$x_\circ$
$\zeta_{\{x_\circ, x_\bullet\}}(x_\bullet)$
$\zeta_{\{x_\circ, x_\bullet\}}(x_\circ)$
G
$\mu_G(\mathbb{S}^2)$

(a) Encoding of the pair (process,view) to a point.

(b) Association between an instant graph of ImS_2 (top) and a simplex of $Chr(\mathbb{S}^2)$ is illustrated.

Figure 2 Construction of $Chr(\mathbb{S}^2)$ as a geometric encoding for IIS_2.

▶ **Definition 31** (Chromatic Subdivision). *Given $(S, \mathcal{P})$ a chromatic simplex, a chromatic subdivision of S is a chromatic simplicial complex $(C, \mathcal{P}_C)$ such that*

- *C is a subdivision of S (*i.e. $\wr C \wr = |S|$*),*
- *$\forall x \in V(S), \mathcal{P}_C(x) = \mathcal{P}(x)$.*

Note that it is not necessary to assume $V(S) \subset V(C)$ here, since the vertices of the simplex S being extremal points, they are necessarily in $V(C)$.

We start by defining some geometric transformations of simplices (here seen as sets of points). The choice of the coefficients will be justified later.

▶ **Definition 32.** *Consider a simplex $V = (y_0, \ldots, y_d)$ of size $d+1$ in $\mathbb{R}^N$. We define the function $\zeta_V : V \longrightarrow \mathbb{R}^N$ by, for all $i \in [0, d]$*

$$\zeta_V(y_i) = \frac{1}{2d+1} y_i + \sum_{j \neq i} \frac{2}{2d+1} y_j$$

We now define directly in a geometric way the *standard chromatic subdivision* of simplex $(S, \mathcal{P})$, where $S = (x_0, x_1, \ldots, x_n)$ and $\mathcal{P}(x_i) = i$.

The chromatic subdivision $Chr(S)$ for the chromatic simplex $S = (x_0, \ldots, x_n)$ is a simplicial complex defined by the set of vertices $V(Chr(S)) = \{\zeta_V(x_i) \mid i \in [0, n], V \subset V(S), x_i \in V\}$.

From the previous definition, for each pair (i, V), $i \in [0, n]$ and $V \subset V(S)$ with $i \in V$, there is an associated vertex $x = \zeta_V(x_i)$ of $Chr(S)$, and conversely each vertex has an associated pair. The *color* of (i, V) is i. The set V is called the *view*. We define Φ the following *presentation* of a vertex x, $\Phi(x) = (\mathcal{P}(x), V_x)$ where $\mathcal{P}(x) = i$ and $V_x = V$.

The simplices of $Chr(S)$ are the set of $d+1$ points $\{\zeta_{V_0}(x_{i_0}), \cdots, \zeta_{V_d}(x_{i_d})\}$ where

- there exists a permutation π on $[0, d]$ such that $V_{\pi(0)} \subseteq \cdots \subseteq V_{\pi(d)}$,
- If $i_j \in \mathcal{P}(V_\ell)$ then $V_j \subset V_\ell$.

In Fig. 2, we present the construction for $Chr(\mathbb{S}^2)$. For convenience, we associate $\circ, \textcolor{gray}{\bullet}, \bullet$ to the processes $0, 1, 2$ respectively. In Fig. 2a, we consider the triangle $x_\circ, x_{\textcolor{gray}{\bullet}}, x_\bullet$ in $\mathbb{R}^2$, with $x_\circ = (0, 0)$, $x_\bullet = (1, 0)$, $x_{\textcolor{gray}{\bullet}} = (\frac{1}{2}, \frac{\sqrt{3}}{2})$. We have that $\zeta_{\{x_\circ, x_\bullet\}}(x_\bullet) = (\frac{1}{3}, 0)$, $\zeta_{\{x_\circ, x_\bullet\}}(x_\circ) = (\frac{2}{3}, 0)$ and $\zeta_{\{x_\circ, x_{\textcolor{gray}{\bullet}}, x_\bullet\}}(x_{\textcolor{gray}{\bullet}}) = (\frac{1}{2}, \frac{\sqrt{3}}{10})$. The relation between instant graph G (top) and simplex $\left\{(\frac{2}{3}, 0), (1, 0), (\frac{1}{2}, \frac{\sqrt{3}}{10})\right\}$ (grey area in Fig. 2b) is detailed in the section 4.2.

In the following, we will be interested in iterations of $Chr(\mathbb{S}^n, \mathcal{P})$. The last property of the definition of chromatic subdivision means with we can drop the C index in the coloring of complex C and use $\mathcal{P}$ to denote the coloring at all steps. From its special role, it is called the *process color* and we sometimes drop $\mathcal{P}$ in $Chr(S, \mathcal{P})$ using in the following $Chr(S)$ for all simplices S of iterations of $Chr(\mathbb{S}^n)$.

In [20], Kozlov showed how the standard chromatic subdivision complex relates to Schlegel diagrams (special projections of cross-polytopes), and used this relation to prove the standard chromatic subdivision was actually a subdivision. In [14, section 3.6.3], a general embedding in $\mathbb{R}^n$ parameterized by $\epsilon \in \mathbb{R}$ is given for the standard chromatic subdivision. The geometrization here is done choosing $\epsilon = \frac{d}{2d+1}$ in order to have "well balanced" drawings.

B.1 Colorless Algorithms in the Iterated Immediate Snapshots Model

It is well known, see [14, Chap. 3&4, Def. 3.6.3], that each maximal simplex $S = \{\zeta_{V_0}(x_{i_0}), \cdots, \zeta_{V_n}(x_{i_n})\}$ from the chromatic subdivision of $\mathbb{S}^n$ can be associated with a graph of ImS_n denoted $\Theta(S)$. In [8], a suitable geometric encoding of the standard chromatic subdivision has been given, this is also detailed here.We can transpose the previous geometric presentation with an averaging algorithm called the *Chromatic Average* Algorithm, presented in Algorithm 1, in a way that encode the IIS model. It was first introduced in [8], here we present the colorless adaptation where only the set of values that is received is taken into account. That is, if two processes send the same value (*i.e.* they are associated to the same point in $\mathbb{R}^N$), this is considered only once in the averaging. Since it still use the formula of 31 this yield again the standard chromatic subdivision.

Executing one round of the loop in Chromatic Average for instant graph G, the state of process i is $x'_i = \zeta_{V_i}(x^*_i)$, where V_i is the view of i on this round, that is the set of (j, x_j) it has received. It use the instant graph of the IIS model that are encoding in the following way : We have $V(\Theta(S)) = \Pi_n = [0, n]$ and set $\Theta(\zeta_{V_j}(x_{i_j})) = \mathcal{P}(x_{i_j})$. The arcs are defined using the representation Φ of points, $A(\Theta(S)) = \{(i, j) \mid i \neq j, V_i \subseteq V_j\}$. The mapping θ will denote Θ^{-1}. Then $\Theta(\{\zeta_{V_0}(x^*_0), \cdots, \zeta_{V_n}(x^*_n)\}) = G$. See eg. in Fig. 2a in the Appendix B. Adjacency for a given i corresponds to the smallest subset containing x_i. This one round transformation for the canonical $\mathbb{S}^n$ can actually be done for any simplex S of dimension n of $\mathbb{R}^N$.

By iterating, the chromatic subdivisions $Chr^r(\mathbb{S}^n)$ are given by the global state under all possible r rounds of the Chromatic Average Algorithm. Finite rounds give the Iterated Chromatic Subdivision (hence the name). This is an algorithm that is not meant to terminate (like the full information protocol). The executions of this algorithm are used below to define a topology on IIS_n.

For $G \in ImS_n$, we denote $\mu_G(S)$ the geometric simplex that is the image of S by one round the Chromatic average algorithm under instant graph G.

To start defining this topology we need to define the function *geo*. Let $w \in IIS_n$, $w = G_1 G_2 \cdots$. For the prefix of w of size r, S a simplex of dimension n, we define $geo(w_{|r})(S) = \mu_{G_r} \circ \mu_{G_{r-1}} \circ \cdots \circ \mu_{G_1}(S)$. Finally, we set $geo(w) = \lim_{r \longrightarrow \infty} geo(w_{|r})$

The Chromatic Average algorithm is therefore the geometric counterpart to the Full Information Protocol that is associated with Chr [14]. In particular, any algorithm can be presented as the Chromatic Average together with a terminating condition and a decision function of x.

C Proof that an IIS-terminating subdivision is a simplicial complex

We will use the following lemma to prove that an IIS-terminating subdivision is a simplicial complex. Note that since we are in the geometric setting, this is not as straightforward as in the abstract setting. We need to carefully check that everything "glues" nicely.

▶ **Lemma 33.** *Let σ a simplex with vertices partitioned in two disjoints set U and T. Then the collection $JOIN(Chr(\sigma(U)), \sigma(T))$ is a simplicial complex.*

Proof. Let $\tau = Chr\,\sigma(U)$ it's a simplicial complex. We have that $|\sigma(U)| = |\tau| \subseteq |\sigma|$. Let $\alpha = JOIN(\tau, \sigma(T))$, the facets of α are the facets of τ in union with the facets of $\sigma(T)$. All off these simplices are closed by inclusion which implies the first property of Def 9. For the intersection property of 9, we take $\beta_1, \beta_2 \in \alpha$ such that $|\beta_1| \cap |\beta_2| = |\beta_3|$ and $|\beta_3| \neq \emptyset$, $\tau_i = \{v \in V(\beta_i) | v \in |\tau|\}$. If $V(\beta_3) \subseteq T$ then β_3 remain unchanged. If $V(\beta_3) = V(\tau_i)$ then because Chr is a subdivision β_3 is a simplicial complex. Else $V(\beta_3)$ is partitioned in $V(\tau)$ and $T(\sigma)$, since τ_i is a subdivision, $JOIN(\tau_i, \sigma(T))$ is a simplicial complex. Moreover, $|\tau_i| \subseteq |\sigma|$ and $|\sigma(T)| \subseteq |\sigma|$, hence β_3 is a simplicial complex. All of this gives that $JOIN(\tau, \sigma(T))$ is indeed a simplicial complex. ◀

▶ **Proposition 34.** *$C = \bigcup C_i(T_i)$ is a simplicial complex.*

For convenience of the reader, we rewrite here the definition of $EChr$:
$EChr(T_i, C_i) = (\bigcup_{\sigma \in C_i} Chr\,\sigma(U_i)) \cup (\bigcup_{\sigma \in C_i} JOIN(Chr\,\sigma(U_i), \sigma(T_i))$.

Proof. We start be proving that for all $i \in \mathbb{N}$, the objects C_i and $C_i(T_i)$ are simplicial complexes.

The first step constructs C_{i+1}, it is a union of two operations. The first one $(\bigcup_{\sigma \in C_i} Chr(\sigma(U_i)))$ takes simplices and apply a mesh-shrinking subdivision, which by definition yields a simplicial complex. The second one $(\bigcup_{\sigma \in C_i} JOIN(V(Chr\,\sigma(U_i), T_\sigma)$ is an union of $JOIN$ on a partition of vertices of a simplex, which by lemma 33 yield again a simplicial complex. We have to prove now that all of this simplices "glues back together nicely". Let $\sigma_1, \sigma_2 \in C_i$ such that $|\sigma_1| \cap |\sigma_2| \neq \emptyset$, then by induction we know that C_i is a simplicial complex then $\exists |\sigma_3| \in C_i, |\sigma_1| \cap |\sigma_2| = |\sigma_3|$. We can make a disjunction of case the vertices of σ_3 to prove that the simplices are intersecting correctly.

1. If $V(\sigma_3) \subseteq T_i$ then the simplex σ_3 is not modified in C_{i+1}
2. If $V(\sigma_3) \subseteq U_i$, the subdivision Chr restricted to α is the same if we look from σ or τ, hence we keep the property of simplicial complexes in C_{i+1}.
3. if $V(\sigma_3) = V(\sigma_3(T_i)) \cup V(\sigma_3(U_i))$ with $V(\sigma_3) \cap T_i \neq \emptyset$ and $V(\sigma_3) \cap U_i \neq \emptyset$. Then by the two later cases, we know that $\sigma_3(T_i)$ and $\sigma_3(U_i)$ preserve the simplicial complex. After that we are doing a $JOIN$ between vertices in the same simplex σ_3 which by lemma 33 yield a simplicial complex.

We can deduce from those 3 cases that C_{i+1} is indeed a simplicial complex, which means that C is also a simplicial complex. ◀

D Additional figure

In Fig. 3 we have $x \in X$, $\mathbf{B}(x, \eta(x))$ is in green. We apply the function f and because it satisfies the η-star condition we can exhibit $v_x \in \mathcal{O}$ such that $f(\mathbf{B}(x, \eta(x) \cap X) \subseteq St^\circ(v_x)$, $St^\circ(v_x))$ is colored in light blue.

The figure 4 outline the relation between the main object of the proof of the proposition 28.

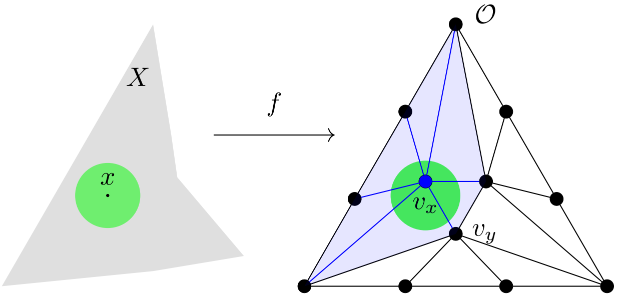


Figure 3 An η-star condition representation.

$$\begin{tikzcd}
Spe_A \arrow[r, "geo"] & E \arrow[d, bend left, "\pi"] \\
& Skel^c\mathcal{I} \arrow[u, bend left, "\pi^{-1}"]
\end{tikzcd}$$

Figure 4 Illustration of proof 28.

E Example of simple fiber bundle and link to distributed system

We acknoledge that fiber bundle might not be a well know mathematical object for some reader, in this section we attempt to adress this difficulty.

One good example of fiber bundle is a Möbius strip. It can be seen as a fiber bundle with a cirlce as B and segment as fiber F. With E the möbius strip, the function $\pi : E \rightarrow B$ is a projection of the segment into the base. It is easy to check that for small portion of the circle there is an homeomorphism to a slice of the möbius strip.

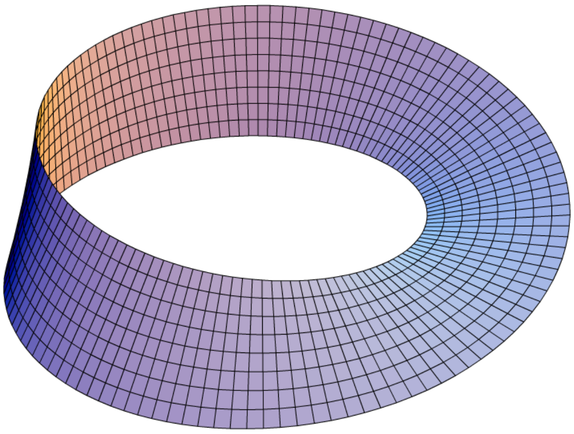


Figure 5 A Möbius Strip.

Breaking Through the $\Omega(n)$-Space Barrier: Population Protocols Decide Double-Exponential Thresholds

Philipp Czerner

Department of Informatics, TU München, Germany

Abstract

Population protocols are a model of distributed computation in which finite-state agents interact randomly in pairs. A protocol decides for any initial configuration whether it satisfies a fixed property, specified as a predicate on the set of configurations. A family of protocols deciding predicates φ_n is *succinct* if it uses $\mathcal{O}(|\varphi_n|)$ states, where φ_n is encoded as quantifier-free Presburger formula with coefficients in binary. (All predicates decidable by population protocols can be encoded in this manner.) While it is known that succinct protocols exist for all predicates, it is open whether protocols with $o(|\varphi_n|)$ states exist for *any* family of predicates φ_n. We answer this affirmatively, by constructing protocols with $\mathcal{O}(\log |\varphi_n|)$ states for some family of threshold predicates $\varphi_n(x) \Leftrightarrow x \geq k_n$, with $k_1, k_2, ... \in \mathbb{N}$. (In other words, protocols with $\mathcal{O}(n)$ states that decide $x \geq k$ for a $k \geq 2^{2^n}$.) This matches a known lower bound. Moreover, our construction for threshold predicates is the first that is not 1-aware, and it is almost self-stabilising.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Distributed computing, population protocols, state complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.17

Related Version *Full Version*: https://arxiv.org/abs/2204.02115 [18]

Funding *Philipp Czerner*: This work was supported by an ERC Advanced Grant (787367: PaVeS) and by the Research Training Network of the Deutsche Forschungsgemeinschaft (DFG) (378803395: ConVeY).

1 Introduction

Population protocols are a distributed model of computation where a large number of indistinguishable finite-state agents interact randomly in pairs. The goal of the computation is to decide whether an initial configuration satisfies a given property. The model was introduced in 2004 by Angluin et al. [4, 5] to model mobile sensor networks with limited computational capabilities (see e.g. [28, 22]). It is also closely related to the model of chemical reaction networks, in which agents, representing discrete molecules, interact stochastically [17].

A protocol is a finite set of transition rules according to which agents interact, but it can be executed on an infinite family of initial configurations. Agents decide collectively whether the initial configuration fulfils some (global) property by *stable consensus*; each agent holds an opinion about the output and may freely change it, but eventually all agents agree.

An example of a property decidable by population protocols is *majority*: initially all agents are in one of two states, x and y, and they try to decide whether x has at least as many agents as y. This property may be expressed by the predicate $\varphi(x, y) \Leftrightarrow x \geq y$.

In a seminal paper, Angluin et al. [7] proved that the predicates that can be decided by population protocols correspond precisely to the properties expressible in Presburger arithmetic, the first-order theory of addition.

To execute a population protocol, the scheduler picks two agents uniformly at random and executes a pairwise transition on these agents. These two agents interact and may change states. The number of agents does not change during the computation. It will be denoted m throughout this paper.

Population protocols are often extended with a *leader* – an auxiliary agent not part of the input, which can assist the computation. It is known that this does not increase the expressive power of the model, i.e. it can still decide precisely the predicates expressible in Presburger arithmetic. However, it is known that leaders enable an exponential speed-up [6, 1] in terms of the time that is needed to come to a consensus.

Space complexity. Many constructions in the literature need a large number of states. We estimate, for example, that the protocols of [6] need tens of thousands of states. This is a major obstacle to implementing these protocols in chemical reactions, as every state corresponds to a chemical compound.

This motivates the study of *space complexity*, the minimal number of states necessary for a population protocol to decide a given predicate. Predicates are usually encoded as quantifier-free Presburger formulae with coefficients in binary. For example, the predicates $\varphi_n(x) \Leftrightarrow x \geq 2^n$ have length $|\varphi_n| \in \Theta(n)$. Formally we define $\text{space}(\varphi)$ as the smallest number of states of any protocol deciding φ, and $\text{space}_L(\varphi)$ as the analogous function for protocols with a leader. Clearly, $\text{space}(\varphi)_L \leq \text{space}(\varphi)$.

The original construction in [4] showed $\text{space}(\varphi) \in \mathcal{O}(2^{|\varphi|})$ – impractically large. For the family of *threshold predicates* $\tau_n(x) \Leftrightarrow x \geq n$ Blondin, Esparza and Jaax [14] prove $\text{space}(\tau_n) \in \mathcal{O}(|\tau_n|)$, i.e. they have polynomial space complexity. For several years it was open whether similarly succinct protocols exist for every predicate. This was answered positively in [13], showing $\text{space}(\varphi) \in \mathcal{O}(\text{poly}(|\varphi|))$ for all φ.

Is it possible to do much better? For most predicates it is not; based on a simple counting argument one can show that for every family φ_n with $|\varphi_n| \in \mathcal{O}(n)$ there is an infinite subfamily $(\varphi'_n)_n \subseteq (\varphi_n)_n$ with $\text{space}_L(\varphi'_n) \in \Omega(|\varphi_n|^{1/4-\varepsilon})$, for any $\varepsilon > 0$ [14].

This covers threshold predicates and many other natural families of protocols (e.g. $\varphi_n(x) \Leftrightarrow x \equiv 0 \pmod n$ or $\varphi_n(x, y) \Leftrightarrow x \geq ny$). But it is not an impenetrable barrier, even for the case of threshold protocols: it does not rule out constructions that work for *infinitely many* (but not all) thresholds and use only, say, logarithmically many states. Indeed, if leaders are allowed this is known to be possible: [14] shows $\text{space}_L(\tau'_n) \in \mathcal{O}(\log|\tau'_n|)$ for some subfamily τ'_n of threshold predicates.

Recently, *general* lower bounds have been obtained, showing $\text{space}(\tau_n) \in \Omega(\log^{1-\varepsilon}|\tau_n|)$ for all $\varepsilon > 0$ [19, 20]. The same bound (up to $\varepsilon = 1/2$) holds even if the model is extended with leaders [24].

For leaderless population protocols, these results leave an exponential gap. In this paper we settle that question and show that, contrary to prevailing opinion, $\text{space}(\tau'_n) \in \mathcal{O}(\log|\tau'_n|)$ for some subfamily τ'_n of threshold predicates. In other words, we construct the first family of *leaderless* population protocols that decide double-exponential thresholds and break through the polynomial barrier.

Robustness. Since population protocols model computations where large numbers of agents interact, it is desirable that protocols deal robustly with noise. In a chemical reaction, for example, there can be trace amounts of unwanted molecules. So the initial configuration of the protocol would have the form $C_I + C_N$, where C_I is the "intended" initial configuration, containing only agents in the designated initial states, and C_N is a "noise" configuration, which can contain agents in arbitrary states.

Table 1 Prior results on the state complexity of threshold predicates $\varphi(x) \Leftrightarrow x \geq k$, for $k \in \mathbb{N}$. Upper bounds need only hold for infinitely many k. We elide exponentially dominated factors from lower bounds.

year	result	type	ordinary	with leaders
2018	Blondin, Esparza, Jaax [14]	construction	$\mathcal{O}(\lvert\varphi\rvert)$	$\mathcal{O}(\log\lvert\varphi\rvert)$
2021	Czerner, Esparza [19]	impossibility	$\Omega(\log\log\lvert\varphi\rvert)$	$\Omega(\text{ack}^{-1}\lvert\varphi\rvert)$
2021	Czerner, Esparza, Leroux [20]	impossibility	$\Omega(\log\lvert\varphi\rvert)$	
2022	Leroux [24]	impossibility		$\Omega(\log\lvert\varphi\rvert)$
2024	this paper	construction	$\mathcal{O}(\log\lvert\varphi\rvert)$	

For threshold predicates, specifically, we want to decide whether $|C_I| + |C_N|$ exceeds some threshold $k \in \mathbb{N}$, under some reasonable restrictions to C_I, C_N. However, all known threshold protocols fail even for the case $|C_N| = 1$. Is it possible to do better?

If C_N can be chosen arbitrarily, then the protocol has to work correctly for *all* input configurations. This property is known as *self-stabilisation*, and it has also been investigated in the context of population protocols [8, 16, 15]. However, it can only be achieved in extensions of the model (e.g. on specific communication graphs, or with a non-constant number of states). This is easy to see in the case of threshold predicates: if any configuration is stably accepting, then any smaller configuration is stably accepting as well. In particular, there is a stably accepting configuration with $k - 1$ agents.

While full self-stabilisation is impossible, in this paper we show that one can come remarkably close. We prove that our construction is *almost self-stabilising*, meaning that it computes the correct output for all C_I, C_N with $|C_I| \geq n$, where n is the number of states of the protocol. We do not constraint C_N at all. Since $n \in \mathcal{O}(\log\log k)$ in our protocol, this means that one can take an arbitrary configuration C_N one wishes to count, add a tiny amount of agents to the initial state, and the protocol will compute the correct output.

Related work. We consider the space complexity of families of protocols, each of which decides a different predicate. In another line of research, one considers a family of protocols for the *same* predicate, where each protocol is specialised for a fixed population size m.

In the original model of population protocols (which is also the model of this paper), the set of states is fixed, and the same protocol can be used for an arbitrary number of agents. Relaxing this requirement has opened up a fruitful avenue of research; here, the number of states depends on m (e.g. the protocol has $\mathcal{O}(\log m)$ states, or even $\mathcal{O}(\log\log m)$ states). In this model, faster protocols can be achieved [3, 26, 27].

It has also led to space-efficient, fast protocols, which stabilise within $\mathcal{O}(\operatorname{polylog} m)$ parallel time, using a state-space that grows only slowly with the number of agents, e.g. $\mathcal{O}(\operatorname{polylog} m)$ states [1, 12, 2, 10, 9, 11, 21]. These protocols have focused on the majority predicate. Moreover, lower bounds and results on time-space tradeoffs have been developed in this model [1, 2].

2 Main result

We construct population protocols (without leaders) for an infinite family of threshold predicates $\varphi_n(x) \Leftrightarrow x \geq k_n$, with $k_1, \ldots \in \mathbb{N}$, proving an $\mathcal{O}(\log|\varphi_n|)$ upper bound on their state complexity. This closes the final gap in the state complexity of threshold predicates.

As in prior work, our result is not a construction for arbitrary thresholds k, only for an infinite family of them. It is, therefore, easier to formally state by fixing the number of states n and specifying the largest threshold k that can be decided by a protocol with n states.

▶ **Theorem 1.** *For every $n \in \mathbb{N}$ there is a population protocol with $\mathcal{O}(n)$ states deciding the predicate $\varphi(x) \Leftrightarrow x \geq k$ for some $k \geq 2^{2^n}$.*

Proof. This will follow from theorems 3 and 5. ◀

The result is surprising, as prevailing opinion was that the existing constructions are optimal. This was based on the following:

- It is intuitive that population protocols with leaders have an advantage. In particular, one can draw a parallel to time complexity, where an exponential gap is proven: for some predicates protocols with leaders have $\mathcal{O}(\operatorname{polylog} m)$ parallel time, while all leaderless protocols have $\Omega(m)$ parallel time.
- The $\mathcal{O}(\log\log k)$-state construction from [14] crucially depends on having leaders.
- The technique to show the $\Omega(\log\log k)$ lower bound could, for the most part, also be used for a $\Omega(\log k)$ bound. Only the use of Rackoff's theorem, a general result for Petri nets, does not extend.
- There is a conditional impossibility result, showing that $\Omega(\log k)$ states are necessary for leaderless 1-aware protocols. [14] (Essentially, protocols where some agent knows at some point that the threshold has been exceeded.) All prior constructions are 1-aware.

Regarding the last point, our protocol evades the mentioned conditional impossibility result by being the first construction that is not 1-aware. Intuitively, our protocol only accepts provisionally and continues to check that no invariant has been violated. Based on this, we also obtain the following robustness guarantee:

▶ **Theorem 2.** *The protocols of Theorem 1 are almost self-stabilising.*

Overview. We build on the technique of Lipton [25], which describes a double-exponential counting routine in vector addition systems. Implementing this technique requires the use of procedure calls; our first contribution are *population programs*, a model in which population protocols can be constructed by writing structured programs, in Section 4. Every such program can be converted into an equivalent population protocol.

However, population programs provide weaker guarantees than the model of parallel programs used in [25]. Both models access registers with values in $\mathbb{N}$. In a parallel program these are initialised to 0, while in a population program *all* registers start with arbitrary values. This limitation is essential for our conversion into population protocols.

A straightforward implementation is, therefore, impossible. Instead, we have to adapt the technique to work with arbitrary initial configurations. Our second contribution, and the main technical difficulty of this result, is extending the original technique with error-checking routines to work in our model. We use a detect-restart loop, which determines whether the initial configuration is "bad" and, if so, restarts with a new initial configuration. The stochastic behaviour of population protocols ensures that a "good" initial configuration is reached eventually. Standard techniques could be used to avoid restarts with high probability and achieve an optimal running time, but this is beyond the scope of this paper.

A high level overview of both the original technique as well as our error-checking strategy is given in Section 5. We then give a detailed description of our construction in Section 6.

To get population protocols, we need to convert from population programs. We split this into two parts. First, we use standard techniques to lower population programs to *population machines*, an assembly-like programming language. In a second step we simulate arbitrary population machines by population protocols. This conversion is described in Section 7.

Finally, we introduce the notion of being almost self-stabilising in Section 8, and prove that our construction has this property.

To start out, Section 3 introduces the necessary mathematical notation and formally defines population protocols as well as the notion of stable computation.

3 Preliminaries

Multisets. We assume $0 \in \mathbb{N}$. For a finite set Q we write $\mathbb{N}^Q$ to denote the set of multisets containing elements in Q. For such a multiset $C \in \mathbb{N}^Q$, we write $C(S) := \sum_{q \in S} C(q)$ to denote the total number of elements in some $S \subseteq Q$, and set $|C| := C(Q)$. Given two multisets $C, C' \in \mathbb{N}^Q$ we write $C \leq C'$ if $C(q) \leq C'(q)$ for all $q \in Q$, and we write $C + C'$ and $C - C'$ for the componentwise sum and difference (the latter only if $C \geq C'$). Abusing notation slightly, we use an element $q \in Q$ to represent the multiset C containing exactly q, i.e. $C(q) = 1$ and $C(r) = 0$ for $r \neq q$.

Stable computation. We are going to give a general definition of stable computation not limited to population protocols, so that we can later reuse it for population programs and population machines. Let $\mathcal{C}$ denote a set of configurations and $\rightarrow$ a left-total binary relation on $\mathcal{C}$ (i.e. for every $C \in \mathcal{C}$ there is a $C' \in \mathcal{C}$ with $C \rightarrow C'$). Further, we assume some notion of output, i.e. some configurations have an output $b \in \{\mathsf{true}, \mathsf{false}\}$ (but not necessarily all).

A sequence $\tau = (C_i)_{i \in \mathbb{N}}$ with $C_i \in \mathcal{C}$ is a *run* if $C_i \rightarrow C_{i+1}$ for all $i \in \mathbb{N}$. We say that τ *stabilises to* b, for $b \in \{\mathsf{true}, \mathsf{false}\}$, if there is an i s.t. C_j has output b for every $j \geq i$. A run τ is *fair* if $\cap_{i \geq 0}\{C_i, C_{i+1}, ...\}$ is closed under $\rightarrow$, i.e. every configuration that *can* be reached infinitely often *is*.

Population protocols. A *population protocol* is a tuple $PP = (Q, \delta, I, O)$, where

- Q is a finite set of *states*,
- $\delta \subseteq Q^4$ is a set of *transitions*,
- $I \subseteq Q$ is a set of *input states*, and
- $O \subseteq Q$ is a set of *accepting states*.

We write transitions as $(q, r \mapsto q', r')$, for $q, r, q', r' \in Q$. A *configuration* of PP is a multiset $C \in \mathbb{N}^Q$ with $|C| > 0$. A configuration C is *initial* if $C(q) = 0$ for $q \notin I$ (one might also say $C \in \mathbb{N}^I$ instead). It has output true if $C(q) = 0$ for $q \notin O$, and output false if $C(q) = 0$ for $q \in O$. For two configurations C, C' we write $C \rightarrow C'$ if $C = C'$ or if there is a transition $(q, r \mapsto q', r') \in \delta$ s.t. $C \geq q + r$ and $C' = C - q - r + q' + r'$.

Let $\varphi : \mathbb{N}^I \rightarrow \{\mathsf{true}, \mathsf{false}\}$ denote a predicate. We say that PP *decides* φ, if every fair run starting at an initial configuration $C \in \mathbb{N}^I$ stabilises to $\varphi(C)$, where fair run and stabilisation are defined as above.

4 Population Programs

We introduce population programs, which allows us to specify population protocols using structured programs. An example is shown in Figure 1.

Formally, a *population program* is a tuple $\mathcal{P} = (Q, \mathrm{Proc})$, where Q is a finite set of *registers* and Proc is a list of *procedures*. Each procedure has a name and consists of (possibly nested) while-loops, if-statements and instructions. These are described in detail below.

```
procedure Main
    $OF$ := false
    while $\neg$Test(4) do
        Clean
    $OF$ := true
    while $\neg$Test(7) do
        Clean
    $OF$ := false
    while true do
        Clean
```

```
procedure Test($i$)
    for $j = 1, ..., i$ do
        if detect $x > 0$ then
            $x \mapsto y$
        else
            return false
    return true
```

```
procedure Clean
    if detect $z > 0$ then
        restart
    swap $x, y$
    while detect $y > 0$ do
        $y \mapsto x$
```

Figure 1 A population program for $\varphi(x) \Leftrightarrow 4 \leq x < 7$ using registers x, y, z. Main is run initially and decides the predicate, Test(i) tries to move i units from x to y and reports whether it succeeded, and Clean checks whether z is empty and moves some number of units from y to x. If Clean detects an agent in z, it restarts the computation. As every run calls Clean infinitely often, this serves to reject initial configurations where z is nonzero; eventually the protocol will be restarted with $z = 0$. This is an illustrative example and some simplifications are possible. E.g. the instruction (**swap** x, y) in Clean is superfluous; additionally, instead of checking $z > 0$ one could omit that register entirely.

Primitives. Each register $x \in Q$ can take values in $\mathbb{N}$. Only three operations on these registers are supported.

- The move instruction ($x \mapsto y$), for $x, y \in Q$, decreases the value of x by one, and increases the value of y by one. We also say that it moves one unit from x to y. If x is empty, i.e. its value is zero, the programs hangs and makes no further progress
- The nondeterministic nonzero-check (**detect** $x > 0$), for $x \in Q$, nondeterministically returns either false or whether $x > 0$. In other words, if it does return true, it certifies that x is nonzero. If it returns false, however, no information has been gained. We consider only fair runs, so if x is nonzero the check cannot return false infinitely often.
- A swap (**swap** x, y) exchanges the values of the two registers x, y. This primitive is not necessary, but it simplifies the implementation.

Loops and branches. Population programs use while-loops and if-statements, which function as one would expect.

We also use for-loops. These, however, are just a macro and expand into multiple copies of their body. For example, in the program in Figure 1 the for-loop in Test expands into i copies of the contained if-statement.

Procedures. Our model has procedure calls, but no recursion. Procedures have no arguments, but we may have parameterised copies of a procedure. The program in Figure 1, for example, has four procedures: Main, Clean, Test(4), and Test(7).

Procedure calls must be acyclic. It is thus not possible for a procedure to call itself, and the size of the call stack remains bounded. We remark that one could inline every procedure call. The main reason to make use of procedures at all is succinctness: if our program contains too many instructions, the resulting population protocol has too many states.

Procedures may return a single boolean value, and procedure calls can be used as expressions in conditions of while- or if-statements.

Output flag. There is an output flag OF, which can be modified only via the instructions OF := true and OF := false. (These are special instructions; it is not possible to assign values to registers.) The output flag determines the output of the computation.

Initialisation and restarts. The only guarantee on the initial configuration is that execution starts at Main. In particular, all registers may have arbitrary values.

There is one final kind of instruction: **restart**. As the name suggests, it restarts the computation. It does so by nondeterministically picking any initial configuration s.t. the sum of all registers does not change.

Size. The *size* of $\mathcal{P}$ is defined as $|Q| + L + S$, where L is the number of instructions and S is the *swap-size*. The latter is defined as the number of pairs $(x, y) \in Q^2$ for which it is syntactically possible for x to swap with y via any sequence of swaps. [1] For example, in Figure 1 the swap-size is two: $(x, y), (y, x)$ can be swapped, but e.g. (x, z) cannot. If we add a (**swap** y, z) instruction at any point, then (x, z) can be swapped (transitively), and the swap-size would be 6.

Configurations and Computation. A *configuration* of $\mathcal{P}$ is a tuple $D = (C, OF, \sigma)$, where $C \in \mathbb{N}^Q$ is the *register configuration*, $OF \in \{\mathsf{true}, \mathsf{false}\}$ is the value of the output flag, and $\sigma \in (\text{Proc} \times \mathbb{N})^*$ is the call stack, storing names and currently executed instructions of called procedures. (E.g. $\sigma = ((\mathsf{Main}, 3), (\mathsf{Test}(4), 1))$ when Test is first called in Figure 1.) A configuration is *initial* if $\sigma = ((\mathsf{Main}, 1))$ and it has *output* OF. For two configurations D, D' we write $D \to D'$ if D can move to D' after executing one instruction.

Using the general notion of stable computation defined in Section 3, we say that $\mathcal{P}$ *decides* a predicate $\varphi(x)$, for $k \in \mathbb{N}$, if every run started at an initial configuration (C, OF, σ) stabilises to $\varphi(|C|)$. Note that this definition limits population programs to decide only unary predicates.

Notation. When analysing population programs it often suffices to consider only the register configuration Let $C, C' \in \mathbb{N}^Q$, $b \in \{\mathsf{false}, \mathsf{true}\}$ and let $f \in \text{Proc}$ denote a procedure. We consider the possible outcomes when executing f in a configuration with registers C. Note that the program is nondeterministic, so multiple outcomes are possible. If f may return b with register configuration C', we write $C, f \to C', b$. For procedures not returning a value, we use $C, f \to C'$ instead. If f may initiate a restart, we write $C, f \to \textbf{restart}$. If f may hang or not terminate, we write $C, f \to \bot$. Finally, we define $\text{post}(C, f) := \{S : C, f \to S\}$.

5 High-level Overview

We give an intuitive explanation of our construction. This section has two parts. As mentioned, we use the technique of Lipton [25] to count to 2^{2^n} using $4n$ registers. We will give a brief explanation of the original technique in Section 5.1. Readers might also find the restatement of Liptons proof in [23] instructive – the Petri net programs introduced therein are closer to our approach, and more similar to models used in the recent Petri net literature.

A straightforward application of the above technique only works if some guarantees are provided for the initial configuration (e.g. that the $4n$ registers used are empty, while an additional register holds all input agents). No such guarantees are given in our model. Instead, we have to deal with adversarial initialisation, i.e. the notion that registers hold arbitrary values in the initial configuration. Section 5.2 describes the problems that arise, as well as our strategies for dealing with them.

[1] Unfortunately, without restrictions we would convert swaps to population protocols with a quadratic blow-up in states, so we introduce this technical notion to quantify the overhead.

5.1 Double-exponential counting

The biggest limitation of population programs is their inability to detect absence of agents. This is reflected in the (**detect** $x > 0$) primitive; it may return true and thereby certify that x is nonzero, but it may always return false, regardless of whether $x = 0$ actually holds. In particular, it is impossible to implement a zero-check.

However, Lipton observes that if we have two registers $x, \overline{x}$ and ensure that the invariant $x + \overline{x} = k$ holds, for some fixed $k \in \mathbb{N}$, then $x = 0$ is equivalent to $\overline{x} \geq k$. Crucially, it is possible to certify the latter property; if we have a procedure for checking $\overline{x} \geq k$, we can run both checks ($x > 0$ and $\overline{x} \geq k$) in a loop until one of them succeeds. Therefore, we may treat x as k-bounded register with deterministic zero-checks.

This seems to present a chicken-and-egg problem: to implement this register we require a procedure for $\overline{x} \geq k$, but checking such a threshold is already the overall goal of the program. Lipton solves this by implementing a bootstrapping sequence. For small k, e.g. $k = 2$, one can easily implement the required $\overline{x} \geq k$ check. We use that as subroutine for *two* k-bounded registers, x and y. Using the deterministic zero-checks, x and y can together simulate a single k^2-bounded register with deterministic zero-check; this then leads to a procedure for checking $\overline{z} \geq k^2$ (for some other register $\overline{z}$).

Lipton iterates this construction n times. We have n levels of registers, with four registers $x_i, y_i, \overline{x}_i, \overline{y}_i$ on each level $i \in \{1, ..., n\}$. For each level we have a constant $N_i \in \mathbb{N}$ and ensure that $x_i + \overline{x}_i = y_i + \overline{y}_i = N_i$ holds. These constants grow by repeated squaring, so e.g. $N_1 = 2$ and $N_{i+1} = N_i^2$. Clearly, $N_n = 2^{2^n}$. (Our actual construction uses slightly different N_i.)

We have not yet broached the topic of initialising these registers s.t. the necessary invariants hold. For our purposes, having a separate initialisation step is superfluous. Instead, we check whether the invariants hold in the initial configuration and restart (nondeterministically choosing a new initial configuration) if they do not.

5.2 Error detection

Our model provides only weak guarantees. In particular, we must deal with adversarial initialisation, meaning that the initial configuration can assign arbitrary values to any register. This is not limited to a designated set of initial registers; all registers used in the computation are affected.

Let us first discuss how the above construction behaves if its invariants are violated. As above, let $x, \overline{x}$ denote registers for which we want to keep the invariant $x + \overline{x} = k$, for some $k \in \mathbb{N}$. If instead $x + \overline{x} > k$, the "zero-check" described above is still guaranteed to terminate, as either $x > 0$ or $\overline{x} \geq k$ must hold. However, it might falsely return $x = 0$ when it is not. The procedure we use above, to combine two k-bounded counter to simulate a k^2-bounded counter, exhibits erratic behaviour under these circumstances. When we try to use it to count to k^2 we might instead only count to some lower value $k' < k^2$, even $k' \in \mathcal{O}(k)$.

If the invariant is violated in the other direction, i.e. $x + \overline{x} < k$ holds, we can never detect $x = 0$ and will instead run into an infinite loop.

The latter case is more problematic, as detecting it would require detecting absence. For the former, we can ensure that we check $x + \overline{x} \geq k + 1$ infinitely often; if $x + \overline{x} > k$, this check will eventually return true and we can initiate a restart. For the $x + \overline{x} > k$ case the crucial insight is that we cannot *detect* it, but we can *exclude* it: we issue a single check $x + \overline{x} \geq k$ in the beginning. If it fails, we restart immediately.

A simplified model. In the full construction, we have many levels of registers that rely on each other. Instead, we first consider a simplified model here to explain the main ideas.

In our simplified model there is only a single register x_i per level $i \in \{1, ..., n\}$ as well as one "level $n+1$" register R. For $i \in \{1, ..., n\}$ we are given subroutines $\textsc{Check}(x_i \geq N_i)$ and $\textsc{Check}(x_i > N_i)$ which we use to check thresholds; however, they are only guaranteed to work if $x_1 = N_1$, $x_2 = N_2$, ..., $x_{i-1} = N_{i-1}$ hold.

Our goal is to decide the threshold predicate $m \geq \sum_i N_i$, where $m := \sum_i x_i + \mathsf{R}$ is the sum of all registers. For each possible value of m we pick one initial configuration C_m and design our procedure s.t.

- every initial configuration different from C_m will cause a restart, and
- if started on C_m it is *possible* that the procedure enters a state where it cannot restart.

The structure of C_m is simple: we pick the largest i s.t. we can set $x_j := N_j$ for $j \leq i$ and put the remaining units into x_{i+1} (or R, if $i = n$). The procedure works as follows:

1. We nondeterministically guess $i \in \{0, ..., n\}$.
2. We run $\textsc{Check}(x_j \geq N_j)$ for all $j \in \{1, ..., i\}$. If one of these checks fails, we restart.
3. According to $i = n$ we set the output flag to `true` or `false`.
4. To verify that we are in C_m, we check the following infinitely often. For $j \in \{1, ..., i\}$ we run $\textsc{Check}(x_j > N_j)$ and restart if it succeeds. If $i < n$ we also restart if $\textsc{Check}(x_{i+1} \geq N_{i+1})$ or one of $x_{i+2}, ..., x_n, \mathsf{R}$ is nonempty.

Clearly, when started in C_m and i is guessed correctly, it is possible for step 2 to succeed, and it is impossible for step 4 to restart. If i is too large, step 2 cannot work, and if i is too small step 4 will detect $x_{i+1} \geq N_{i+1}$. So the procedure will restart until the right i is guessed and step 4 is reached.

Consider an initial configuration $C \neq C_m$, $|C| = m$. There are two cases: either there is a k with $C(x_k) < C_m(x_k)$, or some k has $C(x_k) > C_m(x_k)$. Pick a minimal such k.

In the former case, step 2 can only pass if $i < k$, but then one of $x_{i+2}, ..., x_n, \mathsf{R}$ is nonempty and step 4 will eventually restart.

The latter case is more problematic. Step 2 can pass regardless of i (for $i > k$ the precondition of $\textsc{Check}$ is not met). In step 4, either $i < k$ and then $x_{i+1} \geq N_{i+1}$ or one of $x_{i+2}, ..., x_n, \mathsf{R}$ is nonempty, or $i \geq k$ and one of the checks $\textsc{Check}(x_j > N_j)$ will eventually restart, for $j = k$.

This would be what we are looking for, but note that we implicitly made assumptions about the behaviour of $\textsc{Check}$ when called without its precondition being met. We need two things: all calls to $\textsc{Check}$ terminate and they do not change the values of any register. The second is the simpler one to deal with: later, we will have multiple registers per level and our procedures only need to move agents between registers of the same level. This keeps the sum of registers of one level constant, this weaker property suffices for correctness.

Ensuring that all calls terminate is more difficult. It runs into the problem discussed above, where a zero-check might not terminate if the invariant of its register is violated. In this simplified model it corresponds to the case $x_i < N_i$.

However, we note that $\textsc{Check}(x_i \geq N_i)$ and $\textsc{Check}(x_i > N_i)$ are only called if $(x_1, ..., x_{i-1}) \geq_{\text{lex}} (N_1, ..., N_{i-1})$, where $\geq_{\text{lex}}$ denotes lexicographical ordering. So if the precondition is violated, there must be a $j < i$ with $(x_1, ..., x_{j-1}) = (N_1, ..., N_{j-1})$ and $x_j > N_j$. This can be detected within the execution of $\textsc{Check}$ by calling itself recursively. In this manner, we can implement $\textsc{Check}$ in a way that avoids infinite loops as long as the weaker precondition $(x_1, ..., x_{i-1}) \geq_{\text{lex}} (N_1, ..., N_{i-1})$ holds.

Our actual construction follows the above closely; of course, instead of a single register per level we have four, making the necessary invariants more complicated. Additional issues arise when implementing Check, as registers cannot be detected erroneous while in use. Certain subroutines must hence take care to ensure termination, even when the registers they use are not working properly.

6 A Succinct Population Program

In this section, we construct a population program $\mathcal{P} = (Q, \mathrm{Proc})$ to prove the following:

▶ **Theorem 3.** *Let $n \in \mathbb{N}$. There exists a population program deciding $\varphi(x) \Leftrightarrow x \geq k$ with size $\mathcal{O}(n)$, for some $k \geq 2^{2^{n-1}}$.*

Full proofs and formal definitions of this section can be found in the full version of the paper [18].

We use registers $Q := Q_1 \cup \ldots \cup Q_n \cup \{\mathsf{R}\}$, where $Q_i := \{x_i, y_i, \overline{x}_i, \overline{y}_i\}$ are *level i* registers and R is a *level $n+1$* register. For convenience, we identify $\overline{\overline{x}}$ with x for any register x.

Types of Configurations. As explained in the previous section, x and $\overline{x}$ are supposed to sum to a constant N_i, for a level i register $x \in \{x_i, y_i\}$, which we define via $N_1 := 1$ and $N_{i+1} := (N_i + 1)^2$. If this invariant holds, we can use $x, \overline{x}$ to simulate a N_i-bounded register, which has value x.

We cannot guarantee that this invariant always holds, so our program must deal with configurations that deviate from this. For this purpose, we classify configurations based on which registers fulfil the invariant, and based on the type of deviation.

A configuration $C \in \mathbb{N}^Q$ is *i-proper*, if the invariant holds on levels $1, \ldots, i$, and their simulated registers have value 0. This is a precondition for most routines. Sometimes we relax the latter requirement on the level i registers; C is *weakly i-proper* if it is $(i-1)$-proper and the invariant holds on level i.

If C is $(i-1)$-proper and not i-proper, then there are essentially two possibilities. Either $C \leq C'$ for some i-proper C' and we call C *i-low*, or $C(x) \geq C'$ for a weakly i-proper C' and we call C *i-high*. Note that it is possible that C is neither i-low nor i-high – these configurations are easy to exclude and play only a minor role. We can mostly ensure that i-low configurations do not occur, but procedures must provide guarantees when run on i-high configurations.

Finally, we say that C is *i-empty* if all registers on levels $i, \ldots, n+1$ are empty.

	x_1	$\overline{x}_1$	y_1	$\overline{y}_1$	...	x_{i-1}	$\overline{x}_{i-1}$	y_{i-1}	$\overline{y}_{i-1}$	x_i	$\overline{x}_i$	y_i	$\overline{y}_i$	...
i-proper	0	N_1	0	N_1	...	0	N_{i-1}	0	N_{i-1}	0	N_i	0	N_i	...
weakly i-proper	0	N_1	0	N_1	...	0	N_{i-1}	0	N_{i-1}	3	$N_i - 3$	$N_i - 7$	7	...
i-low	0	N_1	0	N_1	...	0	N_{i-1}	0	N_{i-1}	0	$N_i - 3$	0	N_i	...
i-high	0	N_1	0	N_1	...	0	N_{i-1}	0	N_{i-1}	3	N_i	7	$N_i - 5$	...
i-empty	2	4	8	3	...	5	3	0	7	0	0	0	0	...

Figure 2 Example configurations exhibiting the different types.

Summary. We use the following procedures.

- Main. Computation starts by executing this procedure, and Main ultimately decides the predicate $\varphi(x) \Leftrightarrow x \geq 2\sum_{i=1}^{n} N_i$.

Algorithm AssertEmpty.

```
Parameter: $i \in \{1, ..., n+1\}$
Effect: If $i$-empty, do nothing, else it may restart
procedure AssertEmpty.($i$) [$i \leq n$]
    AssertEmpty($i+1$)
    for $x \in Q_i$ do
        if detect $x > 0$ then
            restart
procedure AssertEmpty.($i$) [$i = n+1$]
    if detect $\mathsf{R} > 0$ then
        restart
```

Algorithm AssertProper.

```
Parameter: $i \in \{1, ..., n\}$
Effect: If $i$-proper or $i$-low, do nothing, else it may restart.
procedure AssertProper.($i$)
    AssertProper($i-1$)
    for $x \in \{x_i, y_i\}$ do
        if detect $x > 0$ then
            restart
        Large($\overline{x}$)
        if detect $x > 0$ then
            restart
```

Algorithm Zero Check whether a register is equal to 0.

```
Parameter: $x \in \{x_i, \overline{x}_i, y_i, \overline{y}_i\}$
Output: whether $x = 0$
procedure Zero($x$)
    while true do
        AssertProper($i-1$)
        if detect $x > 0$ then
            return false
        if Large($\overline{x}$) then
            return true
```

Algorithm IncrPair Decrement a two-digit, base $\beta := N_i + 1$ register.

```
Parameter: $x \in \{x_i, \overline{x}_i\}$, $y \in \{y_i, \overline{y}_i\}$
Effect: $\beta x + y \pmod{\beta^2}$ decreases by 1
procedure IncrPair($x, y$)
    if Zero($\overline{y}$) then
        swap $y, \overline{y}$
        if Zero($\overline{x}$) then
            swap $x, \overline{x}$
        else $\overline{x} \mapsto x$
    else $\overline{y} \mapsto y$
```

- AssertEmpty.. Check whether a configuration is i-empty and initiate a restart if not.
- AssertProper.. Check whether a configuration is i-proper or i-low, initiate a restart if not.
- Large. Nondeterministically check whether a register $x \in Q_i$ is at least N_i.
- Zero. Perform a deterministic zero-check on a register $x \in Q_i$.
- IncrPair. As described in Section 5.1, we use two level i registers (which are N_i bounded) to simulate an N_{i+1}-bounded register. This procedure implements the increment operation for the simulated register.

Procedures AssertEmpty., AssertProper.. The procedure AssertEmpty. is supposed to determine whether a configuration is i-empty, which can easily be done by checking whether the relevant registers are nonempty.

Similarly, AssertProper. is used to ensure that the current configuration is not i-high. If it is, it may initiate a restart. We remark that calls to AssertProper.(0) have no effect and can simply be omitted.

Procedure Zero. This procedure implements a deterministic zero-check, as long as the register configuration is weakly i-proper. To ensure termination, AssertProper. is called within the loop.

Algorithm Large Nondeterministically check whether a register is maximal.

```
Parameter: $x \in \{x_i, \overline{x}_i, y_i, \overline{y}_i\}, x \neq y$
Output: if $x \geq N_i$ return true and swap
    units of $x - N_i$ and $\overline{x}$; or return false
procedure Large($x$)    [for $i = 1$]
    if detect $x > 0$ then
        $x \mapsto \overline{x}$
        swap $x, \overline{x}$
        return true
    else
        return false
procedure Large($x$)    [for $i > 1$]
    if $\neg$Zero($x_{i-1}$) $\vee$ $\neg$Zero($y_{i-1}$) then
        restart
    while true do
        CheckProper($i - 2$)
        if detect $x > 0$ then
            $x \mapsto \overline{x}$
            IncrPair($x_{i-1}, y_{i-1}$)
            if Zero($x_{i-1}$) $\wedge$ Zero($y_{i-1}$) then
                swap $x, \overline{x}$
                return true
        else
            if Zero($x_{i-1}$) $\wedge$ Zero($y_{i-1}$) then
                return false
            if detect $\overline{x} > 0$ then
                $\overline{x} \mapsto x$
                IncrPair($\overline{x}_{i-1}, \overline{y}_{i-1}$)
```

Procedure IncrPair. This is a helper procedure to increment the "virtual", N_{i+1}-bounded counter simulated by x and y. It works by first incrementing the second digit, i.e. y. If an overflow occurs, x is incremented as well. It is also be used to decrement the counter, by running it on $\overline{x}$ and $\overline{y}$.

As we show later, IncrPair is "reversible" under only the weak assumption that the configuration $C \in \mathbb{N}^Q$ is i-high. More precisely, C, IncrPair$(x, y) \rightarrow C'$ implies C', IncrPair$(\overline{x}, \overline{y}) \rightarrow C$. Using this, we can show that Large, which calls IncrPair in a loop, terminates.

Procedure Large. This is the last of the subroutines, and the most involved one. The goal is to determine whether $x \geq N_i$, by using the registers of level $i - 1$ to simulate a "virtual" N_i-bounded register. To ensure termination, we use a "random" walk, which nondeterministically moves either up or down. More concretely, at each step either x is found nonempty, one unit is moved to $\overline{x}$ and the virtual register is incremented, or conversely $\overline{x}$ is nonempty, one unit moved to x, and the virtual register decremented. If the virtual register reaches 0 from above, Large had no effect and returns false. Once the virtual register overflows, a total of N_i units have been moved. These are put back into x by swapping x and $\overline{x}$ and true is returned.

As mentioned above, IncrPair is reversible even under weak assumptions. This ensures that the random walk terminates, as it can always retrace its prior steps to go back to its starting point.

Procedure Main. Finally, we put things together to arrive at the complete program. The implementation is very close to the steps described in Section 5.2 in the simplified model, but instead of guessing an i we iterate through the possibilities.

As mentioned before, Main considers a small set of initial configurations "good" and may stabilise. The following lemma formalises this.

▶ **Lemma 4.** Main, *run on register configuration $C \in \mathbb{N}^Q$, can only restart or stabilise, and*

(a) *it may stabilise to* false *if C is j-low and $(j + 1)$-empty, for some $j \in \{1, ..., n\}$,*

(b) *it may stabilise to* true *if C is n-proper, and*

(c) *it always restarts otherwise.*

Algorithm Main Decide whether there are at least $2\sum_i N_i$ agents.

```
procedure Main
    $OF := \mathsf{false}$
    for $i = 1, ..., n$ do
        while $\neg\mathsf{Large}(\overline{x}_i) \vee \neg\mathsf{Large}(\overline{y}_i)$ do
            $\mathsf{AssertProper}(i)$
            $\mathsf{AssertEmpty}(i+1)$
    $OF := \mathsf{true}$
    while true do
        $\mathsf{AssertProper}(n)$
```

7 Converting Population Programs into Protocols

In the previous section we constructed succinct population programs for the threshold predicate. We now justify our model and prove that we can convert population programs into population protocols, keeping the number of states low. We do this in two steps; first we introduce population machines, which are a low-level representation of population programs, then we convert these into population protocols. This results in the following theorem:

▶ **Theorem 5.** *If a population program deciding φ with size n exists, then there is a population protocol deciding $\varphi'(x) \Leftrightarrow \varphi(x-i) \wedge x \geq i$ with $\mathcal{O}(n)$ states, for an $i \in \mathcal{O}(n)$.*

Population machines are introduced in Section 7.1, they serve to provide a simplified model. Converting population programs into machines is straightforward and uses standard techniques, similar to how one would convert a structured program to use only goto-statements. We will describe this in Section 7.2. The conversion to population protocols is finally described in Section 7.3. Here, we only highlight the key ideas of the conversion. The details can be found in the full version of the paper [18].

7.1 Formal Model

▶ **Definition 6.** *A* population machine *is a tuple $\mathcal{A} = (Q, F, \mathcal{F}, \mathcal{I})$, where Q is a finite set of* registers*, F a finite set of* pointers*, $\mathcal{F} = (\mathcal{F}_i)_{i \in F}$ a list of* pointer domains*, each of which is a nonempty finite set, and $\mathcal{I} = (\mathcal{I}_1, ..., \mathcal{I}_L)$ is a sequence of* instructions*, with $L \in \mathbb{N}$. Additionally, $OF, CF, IP \in F$, $\mathcal{F}_{OF} = \mathcal{F}_{CF} = \{\mathsf{false}, \mathsf{true}\}$ and $\mathcal{F}_{IP} = \{1, .., L\}$. For $x \in Q \cup \{\square\}$ we also require $V_x \in F$, and $x \in \mathcal{F}_{V_x} \subseteq Q$. The* size *of $\mathcal{A}$ is $|Q| + |F| + \sum_{X \in F} |\mathcal{F}_X| + |\mathcal{I}|$.*

Let $x, y \in Q$, $x \neq y$, $X, Y \in F$, $i \in \{1, ..., L\}$ and $f : \mathcal{F}_Y \to \mathcal{F}_X$. There are three types of instructions: $\mathcal{I}_i = (x \mapsto y)$, $\mathcal{I}_i = (\textbf{detect } x > 0)$, or $\mathcal{I}_i = (X := f(Y))$.

A population machine has a number of registers, as usual, and a number of pointers. While each register can take any value in $\mathbb{N}$, a pointer is associated with a finite set of values it may assume. There are three special pointers: the output flag OF, which we have already seen in population programs and is used to indicate the result of the computation, the condition flag CF used to implement branches, and the instruction pointer IP, storing the index of the next instruction to execute. To implement swap instructions we use a register map; the pointer V_x, for a register $x \in Q$, stores the register x is actually referring to. ($V_\square$ is a temporary pointer for swapping.) The model allows for arbitrary additional pointers, we will use a one per procedure to store the return address.

There are only three kinds of instructions: $(x \mapsto y)$ and $(\textbf{detect } x > 0)$ are present in population programs as well and have the same meaning here. (With the slight caveat that x and y are first transformed according to the register map. The instructions do not operate on the actual registers x, y, but on the registers pointed to by V_x and V_y.) The third, $(X := f(Y))$ is a general-purpose instruction for pointers. It can change IP and will be used to implement control flow constructs.

A precise definition of the semantics can be found in the full version of the paper [18].

7.2 From Population Programs to Machines

Population machines do not have high-level constructs such as loops or procedures, but these can be implemented as macros using standard techniques. We show only an example here, a detailed description of the conversion can be found in the full version of the paper [18].

```
procedure Main
    while detect $x > 0$ do
        $x \mapsto y$
    swap $x, y$
```

$\rightsquigarrow$

```
1: detect $x > 0$
2: $IP := \begin{cases} 5 & \text{if } CF \\ 3 & \text{else} \end{cases}$
3: $x \mapsto y$
4: $IP := 1$
5: $V_\square := V_x$
6: $V_x := V_y$
7: $V_y := V_\square$
```

Figure 3 Conversion to a population machine.

Control-flow, i.e. **if**, **while** and procedure calls are implemented via direct assignment to IP, the instruction pointer, as in lines 2 and 4 above. The statements (**detect** $x > 0$) and $(x \mapsto y)$ are translated one-to-one, but note that in the population machine their operands are first translated via the register map. For example, (**detect** $x > 0$) in line 1 checks whether the register pointed to by V_x is nonzero. Correspondingly, **swap** statements result in direct modifications to the register map: lines 5-7 swap the pointers V_x and V_y (and leave the registers they point to unchanged).

7.3 Conversion to Population Protocols

In this section, we only present a simplified version of our construction. In particular, we make use of multiway transitions to have more than two agents interact at a time. Our actual construction, described in the full version of the paper [18], avoids them and the associated overhead.

Let $\mathcal{A} = (Q, F, \mathcal{F}, \mathcal{I})$ denote a population machine. To convert this into a population protocol, we use two types of agents: *register agents* to store the values of the registers, and *pointer agents* to store the pointers. For a register we have many identical agents, and the value of the register corresponds to the total number of those agents. They use states Q. For each pointer we use a unique agent, storing the value of the pointer in its state; they use states $\{X^v : X \in F, v \in \mathcal{F}_X\}$.

Let $X_1, ..., X_{|F|}$ denote some enumeration of F with $X_{|F|} = IP$, and let v_i denote the initial value of X_i. We use X_1 as initial state of the protocol. To goal is to have a unique agent for each pointer, so we implement a simple leader election. We use $*$ as wildcard.

$$X_i^*, X_i^* \mapsto X_i^{v_i}, X_{i+1}^{v_{i+1}} \qquad IP^*, IP^* \mapsto X_1^{v_1}, x$$

with $i \in \{1, ..., |F| - 1\}$. If two agents store the value of a single pointer, they eventually meet and one of them is moved to another state. When this happens, the computation is restarted – but note that the values of the registers are not reset. Eventually, the protocol will thus reach a configuration with exactly one agent in $X_i^{v_i}$, for each i, and the remaining agents in Q.

Starting from this configuration, the instructions can be executed. We illustrate the mapping from instructions to transitions in the following example:

$$\begin{array}{l} 1{:}\;\; x \mapsto y \\ 2{:}\;\; \textbf{detect } x > 0 \\ 3{:}\;\; IP := \begin{cases} 1 & \text{if } CF \\ 4 & \text{else} \end{cases} \\ 4{:}\;\; OF := \neg CF \end{array} \rightsquigarrow \begin{array}{lll} IP^1, V_x^v, V_y^w, v & \mapsto IP^2, V_x^v, V_y^w, w & \text{for } v, w \in Q \\ IP^2, CF^*, V_x^v, v & \mapsto IP^3, CF^{\mathsf{true}}, V_x^v, v & \text{for } v \in Q \\ IP^2, CF^*, V_x^v, w & \mapsto IP^3, CF^{\mathsf{false}}, V_x^v, w & \text{for } w \neq v \\ IP^3, CF^{\mathsf{true}} & \mapsto IP^1, CF^{\mathsf{true}} & \\ IP^3, CF^{\mathsf{false}} & \mapsto IP^4, CF^{\mathsf{false}} & \\ IP^4, OF^*, CF^{\mathsf{true}} & \mapsto IP^5, OF^{\mathsf{false}}, CF^{\mathsf{true}} & \\ IP^4, OF^*, CF^{\mathsf{false}} & \mapsto IP^5, OF^{\mathsf{true}}, CF^{\mathsf{false}} & \end{array}$$

Figure 4 Converting instructions into transitions.

For example, in line 1 we want to move one agent from x to y and set the instruction pointer to 2 (from 1). Recall that the registers map to states of the population protocol via the register map, stored in pointers V_x, where $x \in Q$ is a register. We thus have the following agents initiating the transition:

- IP^1; the agents storing the instruction pointer currently stores the value 1,
- V_x^v; the register $x \in Q$ is currently mapped to state $v \in Q$,
- v; an agent in state v, i.e. representing one unit in register x,
- V_y^w; register y is mapped to state w.

The transition then moves v to state w, and increments the instruction pointer.

The above protocol does not come to a consensus. For this to happen, we use a standard output broadcast: we add a single bit to all states. In this bit an agent stores its current opinion. When any agent meets the pointer agent of the output flag OF, the former will assume the opinion of the latter. Eventually, the value of the output flag has stabilised and will propagate throughout the entire population, at which point a consensus has formed.

8 Robustness of Threshold Protocols

A major motivation behind the construction of succinct protocols for threshold predicates is the application to chemical reactions. In this, as in other environments, computations must be able to deal with errors. Prior research has considered *self-stabilising* protocols [8, 16, 15]. Such a protocol must converge to a desired output regardless of the input configuration. However, it is easy to see that no population protocol for e.g. a threshold predicate can be self-stabilising (and prior research has thus focused on investigating extensions of the population protocol model).

In our definition of population programs, the program cannot rely on any guarantees about its input configuration, so they are self-stabilising by definition. However, when we convert to population protocols, we retain only a slightly weaker property, defined as follows:

▶ **Definition 7.** *Let $PP = (Q, \delta, I, O)$ denote a population protocol deciding φ with $|I| = 1$. We say that PP is* almost self-stabilising*, if every fair run starting at a configuration $C \in \mathbb{N}^Q$ with $C(I) \geq |Q|$ stabilises to $\varphi(|C|)$.*

So the initial configuration can be almost arbitrary, but it must contain a small number of agents in the initial state. In many contexts, this is a mild restriction. In a chemical reaction, for example, the number of agents (i.e. the number of molecules) is many orders of magnitude larger than the number of states (i.e. the number of species of molecules).

In particular, this is also much stronger than any prior construction. All known protocols for threshold predicates are 1-aware [14], and can thus be made to accept by placing a single agent in an accepting state.

▶ **Theorem 2.** *The protocols of Theorem 1 are almost self-stabilising.*

9 Conclusions

We have shown an $\mathcal{O}(\log\log n)$ upper bound on the state complexity of threshold predicates for leaderless population protocols, closing the last remaining gap. Our result is based on a new model, population programs, which enable the specification of leaderless population protocols using structured programs.

As defined, our model of population programs can only decide unary predicates and it seems impossible to decide even quite simple remainder predicates (e.g. “is the total number of agents even”). Is this a fundamental limitation, or simply a shortcoming of our specific choices? We tend towards the latter, and hope that other very succinct constructions for leaderless population protocols can make use of a similar approach.

Our construction is almost self-stabilising, which shows that it is possible to construct protocols that are quite robust against *addition* of agents in arbitrary states. A natural next step would be to investigate the *removal* of agents: can a protocol provide guarantees in the case that a small number of agents disappear during the computation?

Threshold predicates can be considered the most important family for the study of space complexity, as they are the simplest way of encoding a number into the protocol. The precise space complexity of other classes of predicates, however, is still mostly open. The existing results generalise somewhat; the construction presented in this paper, for example, can also be used to decide $\varphi(x) \Leftrightarrow x = k$ for $k \geq 2^{2^n}$ with $\mathcal{O}(n)$ states. As mentioned, there also exist succinct constructions for arbitrary predicates, but – to the extent of our knowledge – it is still open whether, for example, $\varphi(x) \Leftrightarrow x = 0 \pmod{k}$ can be decided for $k \geq 2^{2^n}$, both with and without leaders.

References

1 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L. Rivest. Time-space trade-offs in population protocols. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2560–2579. SIAM, 2017. doi:10.1137/1.9781611974782.169.

2 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2221–2239. SIAM, 2018. doi:10.1137/1.9781611975031.144.

3 Dan Alistarh, Rati Gelashvili, and Milan Vojnovic. Fast and exact majority in population protocols. In Chryssis Georgiou and Paul G. Spirakis, editors, *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 47–56. ACM, 2015. doi:10.1145/2767386.2767429.

4 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *PODC*, pages 290–299. ACM, 2004. doi:10.1145/1011767.1011810.

5 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Comput.*, 18(4):235–253, 2006. doi:10.1007/S00446-005-0138-3.

6 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Comput.*, 21(3):183–199, 2008. doi:10.1007/S00446-008-0067-Z.

7 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Comput.*, 20(4):279–304, 2007. doi:10.1007/S00446-007-0040-2.

8 Dana Angluin, James Aspnes, Michael J. Fischer, and Hong Jiang. Self-stabilizing population protocols. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, volume 3974 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2005. doi:10.1007/11795490_10.

9 Stav Ben-Nun, Tsvi Kopelowitz, Matan Kraus, and Ely Porat. An $O(\log^{3/2} n)$ parallel time population protocol for majority with $O(\log n)$ states. In Yuval Emek and Christian Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 191–199. ACM, 2020. doi:10.1145/3382734.3405747.

10 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. A population protocol for exact majority with o(log5/3 n) stabilization time and theta(log n) states. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 10:1–10:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.DISC.2018.10.

11 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. Time-space trade-offs in population protocols for the majority problem. *Distributed Comput.*, 34(2):91–111, 2021. doi:10.1007/s00446-020-00385-0.

12 Andreas Bilke, Colin Cooper, Robert Elsässer, and Tomasz Radzik. Population protocols for leader election and exact majority with o(logˆ2 n) states and o(logˆ2 n) convergence time. *CoRR*, abs/1705.01146, 2017. arXiv:1705.01146.

13 Michael Blondin, Javier Esparza, Blaise Genest, Martin Helfrich, and Stefan Jaax. Succinct population protocols for Presburger arithmetic. In *STACS*, volume 154 of *LIPIcs*, pages 40:1–40:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.STACS.2020.40.

14 Michael Blondin, Javier Esparza, and Stefan Jaax. Large flocks of small birds: On the minimal size of population protocols. In *STACS*, volume 96 of *LIPIcs*, pages 16:1–16:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.STACS.2018.16.

15 Janna Burman, Ho-Lin Chen, Hsueh-Ping Chen, David Doty, Thomas Nowak, Eric E. Severson, and Chuan Xu. Time-optimal self-stabilizing leader election in population protocols. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 33–44. ACM, 2021. doi:10.1145/3465084.3467898.

16 Shukai Cai, Taisuke Izumi, and Koichi Wada. How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. *Theory Comput. Syst.*, 50(3):433–445, 2012. doi:10.1007/s00224-011-9313-z.

17 Ho-Lin Chen, Rachel Cummings, David Doty, and David Soloveichik. Speed faults in computation by chemical reaction networks. *Distributed Comput.*, 30(5):373–390, 2017. doi:10.1007/s00446-015-0255-6.

18 Philipp Czerner. Breaking through the $\omega(n)$-space barrier: Population protocols decide double-exponential thresholds, 2024. arXiv:2204.02115.

19 Philipp Czerner and Javier Esparza. Lower bounds on the state complexity of population protocols. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 45–54. ACM, 2021. doi:10.1145/3465084.3467912.

20 Philipp Czerner, Javier Esparza, and Jérôme Leroux. Lower bounds on the state complexity of population protocols. *CoRR*, 2021. doi:10.48550/arXiv.2102.11619.

21 David Doty, Mahsa Eftekhari, Leszek Gasieniec, Eric E. Severson, Przemyslaw Uznanski, and Grzegorz Stachowiak. A time and space optimal stable population protocol solving exact majority. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1044–1055. IEEE, 2021. doi:10.1109/FOCS52979.2021.00104.

22 Moez Draief and Milan Vojnovic. Convergence speed of binary interval consensus. *SIAM J. Control. Optim.*, 50(3):1087–1109, 2012. doi:10.1137/110823018.

23 Javier Esparza. Decidability and complexity of petri net problems - an introduction. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 374–428. Springer, 1996. doi:10.1007/3-540-65306-6_20.

24 Jérôme Leroux. State complexity of protocols with leaders. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 257–264. ACM, 2022. doi:10.1145/3519270.3538421.

25 Richard J. Lipton. The reachability problem requires exponential space. Technical report, Yale University, Dept. of CS, 1976. URL: http://www.cs.yale.edu/publications/techreports/tr63.pdf.

26 Yves Mocquard, Emmanuelle Anceaume, James Aspnes, Yann Busnel, and Bruno Sericola. Counting with population protocols. In D. R. Avresky and Yann Busnel, editors, *14th IEEE International Symposium on Network Computing and Applications, NCA 2015, Cambridge, MA, USA, September 28-30, 2015*, pages 35–42. IEEE Computer Society, 2015. doi:10.1109/NCA.2015.35.

27 Yves Mocquard, Emmanuelle Anceaume, and Bruno Sericola. Optimal proportion computation with population protocols. In Alessandro Pellegrini, Aris Gkoulalas-Divanis, Pierangelo di Sanzo, and Dimiter R. Avresky, editors, *15th IEEE International Symposium on Network Computing and Applications, NCA 2016, Cambridge, Boston, MA, USA, October 31 - November 2, 2016*, pages 216–223. IEEE Computer Society, 2016. doi:10.1109/NCA.2016.7778621.

28 Etienne Perron, Dinkar Vasudevan, and Milan Vojnovic. Using three states for binary consensus on complete graphs. In *INFOCOM 2009. 28th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro, Brazil*, pages 2527–2535. IEEE, 2009. doi:10.1109/INFCOM.2009.5062181.

On the Limits of Information Spread by Memory-Less Agents

Niccolò D'Archivio

COATI, INRIA d'Université Côte d'Azur, Sophia-Antipolis, France

Robin Vacus

Bocconi Institute for Data Science and Analytics, Bocconi University, Milan, Italy

Abstract

We address the self-stabilizing bit-dissemination problem, designed to capture the challenges of spreading information and reaching consensus among entities with minimal cognitive and communication capacities. Specifically, a group of n agents is required to adopt the correct opinion, initially held by a single informed individual, choosing from two possible opinions. In order to make decisions, agents are restricted to observing the opinions of a few randomly sampled agents, and lack the ability to communicate further and to identify the informed individual. Additionally, agents cannot retain any information from one round to the next. According to a recent publication by Becchetti et al. in SODA (2024), a logarithmic convergence time without memory is achievable in the parallel setting (where agents are updated simultaneously), as long as the number of samples is at least $\Omega(\sqrt{n \log n})$. However, determining the minimal sample size for an efficient protocol to exist remains a challenging open question. As a preliminary step towards an answer, we establish the first lower bound for this problem in the parallel setting. Specifically, we demonstrate that it is impossible for any memory-less protocol with constant sample size, to converge with high probability in less than an almost-linear number of rounds. This lower bound holds even when agents are aware of both the exact value of n and their own opinion, and encompasses various simple existing dynamics designed to achieve consensus. Beyond the bit-dissemination problem, our result sheds light on the convergence time of the "minority" dynamics, the counterpart of the well-known majority rule, whose chaotic behavior is yet to be fully understood despite the apparent simplicity of the algorithm.

2012 ACM Subject Classification Theory of computation → Random walks and Markov chains; Theory of computation → Distributed algorithms

Keywords and phrases Opinion Dynamics, Consensus Protocols, Collective Animal Behavior, Lower Bound

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.18

Related Version *Full Version*: `https://arxiv.org/abs/2402.11553`

Funding *Niccolò D'Archivio*: This work has been supported by the AID INRIA-DGA project n°2023000872 "BioSwarm".

Acknowledgements The authors would like to thank Andrea Clementi for his feedback, and Amos Korman for helpful comments and coming up with the minority dynamics. Thanks also to Luca Trevisan for preliminary discussions, and for everything else.

1 Introduction

Exploring the computational power and limits of well-chosen models – ones that are simple enough for analytical tractability, and yet relevant to specific biological scenarios – can lead to insightful conclusion about the functioning of biological distributed systems [1, 18, 16, 20, 11]. In line with this approach, we consider the *bit-dissemination* problem, introduced in [9] in order to evaluate the possibility to solve two fundamental problems concurrently: reaching agreement efficiently, while ensuring that an information possessed initially by a single individual is propagated to the whole group. In order to fit biological scenarios, the problem features extremely constrained communications. Agents engage in random interactions with just a few individuals at a time, as in the $\mathcal{PULL}$ model. Furthermore, they can only disclose their current decision and no other information, following an assumption introduced in [23] to model situations in which individuals do not actively communicate [14].

In this paper, we further restrict attention to *memory-less* entities lacking the ability to perform computations over extended periods of time; or at least not in a sufficiently reliable manner. In particular, this assumption precludes the possibility to maintain clocks and counters, or to estimate the tendency of the dynamics. Biological ensembles for which it is plausible include ant colonies [17], slime molds, cells and bacteria [24] or even plants [29].

Although it is always hard to rule out the possibility that species make use of memory (especially in the case of social insects), this modeling choice remains applicable even without presupposing specific cognitive abilities, as many species often stick to simple behavioral rules when attempting to reach a consensus, such as quorum sensing [28] or alignment rules [30]. Furthermore, our goal is not to focus on one truly realistic model, but rather to find out what are the minimal requirements to perform certain tasks.

Informal description of the problem

More precisely, we consider a group of n agents holding binary opinions. One of the agents, referred to as the *source*, knows what opinion is "correct" and remains with it at all times. Execution proceeds in discrete rounds. We assume that agents have no memory of what happened in previous rounds, besides their current opinion. In the *parallel* setting, all non-source agents are activated simultaneously in every round, while in the *sequential* setting, only one non-source agent, selected uniformly at random, is activated. Upon activation, a non-source agent i samples a set S consisting of ℓ other agents drawn uniformly at random (with replacement[1]). Then, based only on the opinions of the agents in S and on its own opinion, Agent i may choose to adopt a new opinion. In particular, Agent i does not know whether or not S contains the source. A protocol is successful if every non-source agent eventually adopts the correct opinion and remains with it forever. Finally, a protocol must converge independently of the initial opinions of the agents (including the correct opinion), which can be thought of as being chosen by an adversary.

Previous works

The main parameters of the problems are the activation pattern (which may be parallel or sequential) and the sample size ℓ. To compare protocols across various settings, we are typically interested in their *convergence time*, i.e., the total number of activations required to reach consensus w.r.t. the correct opinion. Here, we will express the convergence time in terms of *parallel rounds*: one parallel round is made up of n activations, which corresponds to 1 round in the parallel setting and n rounds in the sequential setting (although the two settings are not equivalent).

The bit-dissemination problem without memory was first studied in [5], in the sequential setting, where nearly matching lower and upper bounds are given. On the one hand, the authors show that no protocol can converge in less than $\Omega(n)$ parallel rounds in expectation, regardless of the sample size. On the other hand, they show that the well-known Voter

[1] Note that when $\ell \ll n$, sampling with and without replacement are essentially equivalent, and we choose the former for the sake of convenience.

dynamics (Algorithm 1 in Appendix A) achieves consensus in $O(n \log^2 n)$ parallel rounds with high probability. Since the Voter dynamics only needs the sample size to be 1, these results imply that ℓ is not a critical parameter in the sequential setting.

Later, the authors of [7] show that the aforementioned lower bound does not hold in the parallel setting. In fact, they show that the *minority* dynamics (Algorithm 2 in Appendix A) converges in $O(\log^2 n)$ parallel rounds w.h.p., as long as the sample size is at least $\Omega(\sqrt{n \log n})$. This finding reveals that the best convergence times achievable in the sequential versus parallel settings differ by an exponential factor. The bit-dissemination problem is all the more interesting to study as it is one of the most natural ones exhibiting this property.

Part of the explanation behind this curious phenomenon is that the stochastic processes involved are of different mathematical natures, in one setting compared to the other. In the sequential setting, the number of agents with opinion 1 may only vary by at most one unit in every round, since only one agent is activated at a time. Therefore, independently of the protocol being operated, the evolution of the system can always be described by a "birth-death" chain, i.e., a Markov chain whose underlying graph is a path of size n. In fact, all proofs in [5] heavily rely on this observation. In contrast, in the parallel setting, the process may jump from any configuration to any other – albeit with extremely small probability. On the one hand, this characteristic allows for fast convergence. For instance, qualitatively speaking, the minority dynamics succeeds by first reaching a configuration in which an appropriate proportion of the agents hold the wrong opinion; after what all non-source agents, perceiving the same minority, simultaneously adopt the correct opinion. On the other hand, it complicates the analysis of any protocol in the parallel setting, and even more so the task of deriving lower bounds. Following on from these works, we are ultimately interested in the following question:

Is there any protocol achieving a poly-logarithmic convergence time in the parallel setting, when the sample size is $o(\sqrt{n})$?

Based on the findings in [7], the minority dynamics is a natural candidate for this task. Despite its extreme simplicity, the conditions under which it is able to converge quickly have not been identified. While the analysis in [7] relies on a sample size of at least $\Omega(\sqrt{n \log n})$, the authors do not provide a lower bound for this parameter, nor do they justify informally why this quantity is necessary. Therefore, we also consider the following independent question.

What is the minimal sample size for which the minority dynamics converges in poly-logarithmic time?

As a first step towards answering these questions, we focus on the case that the sample size ℓ is constant, or in other words, independent of n. Beyond analytical tractability, several reasons motivate this assumption. First, many existing opinion dynamics that have been traditionally studied in the context of consensus are defined with a small (fixed) number of samples [6]. This is the case for the Voter dynamics and majority dynamics, but also for the undecided states dynamics, as well as all *population protocols* [2], in which agents interact by pairs. In addition, protocols relying on a sample size that increases with n implicitly require the agents to have some knowledge about the size of the population, which is often undesirable or unrealistic in distributed systems. Finally, real biological entities are most likely interacting with few of their conspecifics even when they are part of a larger group. As an illustration, it has been shown empirically that the movement of any bird in a flock depends mostly on its 6 or 7 closest neighbors, regardless of how many individuals are present in the vicinity [3, 8], while other works point at a similar phenomenon in fishes [22].

1.1 Problem Definition

We consider a finite set $I = \{1, \dots, n\}$ of agents. Let $X_t^{(i)} \in \{0, 1\}$ be the *opinion* of Agent i in round t. We assume that Agent 1 is the *source* and holds the *correct* opinion throughout the execution. Denoting the sample size by ℓ (independent of n), a protocol $\mathcal{P}$ is defined as a family of functions

$$g_n^{[b]} : \{0, \dots, \ell\} \to [0, 1].$$

For a given population size n, an opinion b, and a sample of opinions containing k '1' out of ℓ, $g_n^{[b]}(k)$ gives the probability that an agent adopts opinion 1 in the next round when operating the corresponding protocol. Specifically, the process is obtained by performing the following two steps for every agent i simultaneously, in every round t:

1. A vector $S_t^{(i)} \in I^\ell$ of size ℓ is sampled uniformly at random (u.a.r.) (the same agent may appear several times in $S_t^{(i)}$, and i may appear in it as well).
2. Writing $k_t^{(i)}$ to denote the number of agents with opinion 1 in $S_t^{(i)}$, Agent i updates its opinion according to

$$X_{t+1}^{(i)} \leftarrow 1 \text{ with probability } g_n^{\left[X_t^{(i)}\right]}(k_t^{(i)}), \quad 0 \text{ otherwise.}$$

For the sake of clarity, we note a few important consequences of this definition.

- Non-source agents do not know where the opinions that they observe come from. In particular, they do not know if $S_t^{(i)}$ contains the source.
- Non-source agents do not have identifiers, or in other words, all of them must run exactly the same update rule. They are also not aware of the round number (indices are used for analysis purposes only).
- However, non-source agents are aware of their current opinion, as well as the exact value of n.
- Besides their opinion, non-source agents have no memory, in the sense that their behavior cannot depend on any information from previous rounds.

Since agents have no memory besides their opinion, and no identifiers, the configuration of the system in round t can be described simply by a pair (z, X_t), where $z \in \{0, 1\}$ denotes the correct opinion, and $X_t \in \{0, \dots, n\}$ denotes the number of agents with opinion 1. For a given $n \in \mathbb{N}$, and an initial configuration $C = (z, X_0)$, we define the convergence time of protocol $\mathcal{P}$ as the first round for which all agents have adopted the correct opinion and remain with it forever, that is:

$$\tau_n(\mathcal{P}, C) := \inf\{t \geq 0, \text{ for every } s \geq t, X_s = n \cdot z\}.$$

Given a sequence of events $\{A_n\}_{n\in\mathbb{N}}$, we say that "A_n happens with high probability (w.h.p.)" if $\mathbb{P}(A_n) = 1 - 1/n^{\Omega(1)}$.

1.2 Our Results

We show that, when the sample size is bounded, any protocol that does not have access to memory needs almost-linear time to solve the bit-dissemination problem.

▶ **Theorem 1.** *Assume that the sample size ℓ is constant. For every protocol $\mathcal{P}$, there exists a sequence of initial configuration C_n such that for every $\varepsilon > 0$, the convergence time of $\mathcal{P}$ is greater than $n^{1-\varepsilon}$ w.h.p.:*

$$\mathbb{P}\big[\tau_n(\mathcal{P}, C_n) < n^{1-\varepsilon}\big] = \frac{1}{n^{\Omega(1)}}.$$

To the best of our knowledge, this is the first non-trivial lower-bound for this problem in the parallel setting. The proof of Theorem 1 is presented in Section 4, and uses a general result on Markov chains, described in Section 3, as a black box. It consists in studying a kind of "characteristic" function, defined as

$$F_n(p) := -p + \sum_{k=0}^{\ell} \binom{\ell}{k} p^k (1-p)^{\ell-k} \left(p\, g_n^{[1]}(k) + (1-p)\, g_n^{[0]}(k) \right). \tag{1}$$

The sum in Equation (1) corresponds to the probability that a non-source agent, taken uniformly at random, adopts opinion 1, given that the current proportion of agents with opinion 1 is p. Informally, $F_n(p)$ measures the "bias" of a protocol $\mathcal{P}$ towards opinion 1, or in other words:

$$\mathbf{E}\left(\frac{X_{t+1}}{n} \mid X_t = x_t \right) \approx \frac{x_t}{n} + F_n\left(\frac{x_t}{n} \right)$$

(see Proposition 5 for a more accurate statement). As a consequence, the sign of the function at p provides information on the trend of the dynamics when the proportion of agents with opinion 1 is p; moreover, roots correspond to fixed points of the dynamics (either stable or unstable).

Similar functions have already been defined in the literature, where they are typically used to obtain two types of results. On the one hand, they can be used to identify phase transitions [12, 13]. For example, when such characteristic function depends on a parameter α, there might be a critical value $\alpha^\star$ at which a new root appears. In that case, the behavior of the dynamics below and above $\alpha^\star$ can be significantly different. On the other hand, if a characteristic function has a constant sign over a large interval (and under some additional conditions), the dynamics cannot easily travel in the opposite direction, which can be exploited to obtain lower bounds on the convergence time [12].

In this paper, we use function F_n yet in another way. Specifically, we leverage the fact that a bound on the sample size ℓ implies a bound on the degree of F_n, and therefore a bound on the number of roots within the interval $[0,1]$. Then, we consider a well-chosen interval of constant length between two roots of F_n, and employ the aforementioned argument to obtain a lower bound: if F_n is negative on this interval, i.e., $\mathcal{P}$ tends to make the proportion of 1-opinions decrease, we show that the process will be slow to reach consensus every time the correct opinion is 1. Conversely, if it is positive, we show that fast convergence fails whenever the correct opinion is 0.

In terms of the dependency on n, we show that our lower bound is nearly tight (up to a sub-polynomial factor) by adapting a well-known result to our setting. Its proof does not introduce any novel argument and is deferred to Appendix C.

▶ **Theorem 2.** *Consider the Voter dynamics $\mathcal{P}^{voter}$, with sample size $\ell = 1$. For every sequence of initial configuration C_n, the convergence time of $\mathcal{P}^{voter}$ is less than $2n \log n$ w.h.p.:*

$$\mathbb{P}\left[\tau_n(\mathcal{P}, C_n) \leq 2n \log n\right] \geq 1 - \frac{1}{n^2}.$$

When it comes to parameter ℓ, a gap remains between our lower bound and the upper bound in [7], where it is shown that the minority dynamics solves the problem in $O(\log^2 n)$ rounds w.h.p. when ℓ is at least $\Omega(\sqrt{n \log n})$.

Unfortunately, we believe that our techniques cannot be used to extend the lower bound to a higher value of ℓ. Indeed, if $\ell = \Omega(\log n)$, it is already possible for a protocol to converge in just one round w.h.p. from configurations that are arbitrarily far away from the consensus.

This observation destroys any hope of restricting the analysis to a small interval of the configuration space. In contrast, this phenomenon does not happen w.h.p. in our setting (see Proposition 4). However, we have no good reason to think that $\Theta(\sqrt{n \log n})$ is the smallest value of ℓ allowing for an efficient protocol (such as the minority dynamics) – this is left as an open problem.

1.3 Other Related Works

The bit-dissemination problem was also studied under the assumption that agents can use a moderate amount of memory. An efficient protocol is identified in [23], and achieves consensus in $O(\text{polylog } n)$ parallel rounds with high probability. It relies on agents being able to memorize $\log \log n$ bits of information from one round to the next and requires a sample size logarithmic in n. Other candidates are mentioned in [9] but are not analysed. The authors of [15] show that the problem can be solved in the context of population protocols with a memory of only constant size. Importantly however, population protocols do not fit the framework of passive communications. Specifically, interaction rules in this model depend on the exact states of the agents, and not just on their binary opinion.

The bit-dissemination problem is a specific case of the *majority bit-dissemination* problem, introduced in [9] and also addressed in [15]. In this variant, the number of source agents is arbitrarily large, and they may have conflicting *preferences*. The opinions of sources must not necessarily be in line with their preferences, and they can participate to the protocol in the same way as regular agents. The correct opinion is defined as the most widespread preference among sources. On the one hand, an efficient solution can be derived from the results in [15, 4], but require active communications, and relies on memory. On the other hand, the authors of [23] show that the majority bit-dissemination problem is impossible with passive communications.

More generally, many works within the opinion dynamics literature investigate the influence of the presence of “stubborn” or “biased” agents on the behaviour of the system. Typically, these works focus on a single arbitrary process, mainly the Voter dynamics [27, 31, 19, 26, 25], and to the best of our knowledge, do not establish general lower bounds. In contrast, our goal is to better understand the difficulty of spreading information as an algorithmic problem; therefore, we do not want to rule out any imaginable protocol within the constraints of our setting. Furthermore, they often investigate different questions, such as the impact of the number of sources or they position in the network on the convergence time, or assume that sources may have conflicting opinions.

2 Preliminaries

In this section, we make a few general observations that we will use later in our analysis. We assume that ℓ is a constant w.r.t. n.

Conditioning on Agent i sampling exactly k times the opinion “1”, for every $k \in \{0, \dots, \ell\}$, we obtain that

$$\mathbb{P}\left(X_{t+1}^{(i)} = 1 \mid X_t = x_t,\ X_t^{(i)} = b\right) = \sum_{k=0}^{\ell} \binom{\ell}{k} \left(\frac{x_t}{n}\right)^k \left(1 - \frac{x_t}{n}\right)^{\ell-k} g_n^{[b]}(k). \tag{2}$$

After convergence has happened, i.e., $X_t \in \{0, n\}$, this probability must be equal to 0 or 1 respectively so that a consensus is maintained. This imposes a constraint on any protocol attempting to solve the bit-dissemination problem, which can be formalized as follows.

▶ **Proposition 3.** *Any protocol $\mathcal{P}$ solving the bit-dissemination problem must satisfy $g_n^{[0]}(0) = 0$ and $g_n^{[1]}(\ell) = 1$.*

Proof. Consider the case that the correct opinion $z = 0$. If $X_t = 0$ for some round t, then each agent has opinion 0, and receives exactly 0 samples equal to '1'. Following the protocol, all agents adopt opinion 1 in the next round w.p. $g_n^{[0]}(0)$ independently of each-other. If $g_n^{[0]}(0) > 0$, then

$$\mathbb{P}\left(X_{t+1} = 0 \mid X_t = 0\right) = \prod_{i \geq 2} \mathbb{P}\left(X_{t+1}^{(i)} = 0 \mid X_t = 0\right) = \left(1 - g_n^{[0]}(0)\right)^{n-1} < 1.$$

Therefore, $\inf\{t \geq 0,\ \text{for every } s \geq t, X_s = 0\} = +\infty$ almost surely, and the protocol cannot solve the bit-dissemination problem in the sense of Section 1.1. By symmetry, we obtain the other statement about $g_n^{[1]}(\ell)$, which concludes the proof of Proposition 3. ◀

Accordingly, we will always assume that $g_n^{[0]}(0) = 0$ and $g_n^{[1]}(\ell) = 1$. Using this assumption, we can show a general upper bound on the fraction of agents with opinion 0 that can change opinion in a single time step.

▶ **Proposition 4.** *Let $c \in (0,1)$ and consider a protocol $\mathcal{P}$ solving the bit-dissemination problem. There is a constant $y = y(c,\ell) \in (c,1)$ s.t. for every n large enough, and $x_t \leq c\,n$,*

$$\mathbb{P}\left(X_{t+1} \leq y\,n \mid X_t = x_t\right) \geq 1 - \exp\left(-2\,n^{-1/2}\right).$$

Proof. Let $t \in \mathbb{N}$, and $x_t \leq cn$. By Equation (2), and since we assumed $g_n^{[0]}(0) = 0$,

$$\begin{aligned}
\mathbb{P}\left(X_{t+1}^{(i)} = 0 \mid X_t = x_t, X_t^{(i)} = 0\right) &= 1 - \sum_{k=0}^{\ell} \binom{\ell}{k} \left(\frac{x_t}{n}\right)^k \left(1 - \frac{x_t}{n}\right)^{\ell-k} g_n^{[0]}(k) \\
&= \sum_{k=0}^{\ell} \binom{\ell}{k} \left(\frac{x_t}{n}\right)^k \left(1 - \frac{x_t}{n}\right)^{\ell-k} \left(1 - g_n^{[0]}(k)\right) \\
&\geq \left(1 - \frac{x_t}{n}\right)^{\ell} \geq (1-c)^{\ell}.
\end{aligned}$$

Let Y be the number of agents with opinion 0 in round t, that keep opinion 0 in round $t+1$ (conditioning on $X_t = x_t$). By assumption, $n - X_t \geq (1-c)n$, so the last equation implies the following domination[2]:

$$Y \succeq \text{Binomial}\left((1-c)n, (1-c)^{\ell}\right) := Z.$$

Let $a = a(c,\ell) := (1-c)^{\ell+1}$, so that $\mathbf{E}(Z) = an$. Let $a' := a - n^{-1/4}$. For n large enough, we have $a' > a/2$. By Hoeffding's bound, we have

$$\mathbb{P}\left(Z \leq \frac{an}{2}\right) \leq \mathbb{P}\left(Z \leq a'n\right) = \mathbb{P}\left(Z \leq \mathbf{E}(Z) - n^{3/4}\right) \leq \exp\left(-2n^{1/2}\right).$$

Finally, setting $y = y(c,\ell) := \max(1 - a/2,\ c)$ (so that $y \in (c,1)$), we obtain

$$\mathbb{P}\left(X_{t+1} \geq y\,n \mid X_t = x_t\right) \leq \mathbb{P}\left(Y \leq \frac{an}{2}\right) \leq \mathbb{P}\left(Z \leq \frac{an}{2}\right) \leq \exp\left(-2\,n^{1/2}\right),$$

which concludes the proof of Proposition 4. ◀

[2] Given two real-valued random variables X and Y, we say that X is *stochastically dominated* by Y, and write $X \preceq Y$, if for every $x \in \mathbb{R}$, $\mathbb{P}(X > x) \leq \mathbb{P}(Y > x)$.

Finally, the following proposition justifies the informal claim made in Section 1.2 according to which the function F_n, defined in Equation (1), represents the "bias" of the corresponding protocol towards opinion 1.

▶ **Proposition 5.** *For every protocol $\mathcal{P}$, and every $x_t \in [n]$,*

$$\mathbf{E}\left(X_{t+1} \mid X_t = x_t\right) \leq x_t + n\, F_n\left(\frac{x_t}{n}\right) + 1, \tag{3}$$

$$\mathbf{E}\left(X_{t+1} \mid X_t = x_t\right) \geq x_t + n\, F_n\left(\frac{x_t}{n}\right) - 1. \tag{4}$$

Proof. For a non-source agent $i \in I \setminus \{1\}$, an opinion $b \in \{0,1\}$, and any $p \in [0,1]$, let

$$P_b := \mathbb{P}\left(X_{t+1}^{(i)} = 1 \mid X_t = np,\ X_t^{(i)} = b\right) = \sum_{k=0}^{\ell} \binom{\ell}{k} p^k (1-p)^{\ell-k} g_n^{[b]}(k),$$

where the second equality is a restatement of Equation (2). Note that by definition of F_n,

$$F_n(p) = p\,P_1 + (1-p)P_0 - p. \tag{5}$$

Denoting by z the correct opinion, one can check that there are $X_t - z$ non-source agents with opinion 1 in round t, and $n - X_t - (1-z)$ non-source agents with opinion 0. Hence,

$$\begin{aligned}\mathbf{E}\left(X_{t+1} \mid X_t = np\right) &= z + (np - z)P_1 + \big(n - np - (1-z)\big)P_0 \\ &= n\big(pP_1 + (1-p)P_0\big) + z(1-P_1) - (1-z)P_0 \\ &= np + n\,F_n(p) + z(1-P_1) - (1-z)P_0. \qquad \text{(by Equation (5))}\end{aligned}$$

Note that for any source opinion $z \in \{0,1\}$, since $P_0, P_1 \in [0,1]$, we have

$$-1 \leq z(1-P_1) - (1-z)P_0 \leq +1,$$

from which Equations (3) and (4) follow by taking $p := x_t/n$. ◀

3 An Intermediate Result on Markov Chains

In this section, we present a result that we will use later as a black box (with $a_1, a_2, a_3 \in [0,1]$). Informally, the theorem says that if a Markov chain is a super-martingale over an interval of values, and given that it cannot skip the interval entirely, then the time required to cross the interval is at least the time needed by a martingale to escape it.

▶ **Theorem 6.** *Let $\{X_t\}_{t\in\mathbb{N}}$ be a Markov chain on $\mathbb{Z}$ and $\varepsilon > 0$. If there are $a_1 < a_2 < a_3 \in \mathbb{R}$ s.t.*

(i) *for every $x_t \in \{\lceil a_1\, n\rceil, ..., \lfloor a_3\, n\rfloor\}$, $\mathbf{E}(X_{t+1} \mid X_t = x_t) \leq x_t + 1$,*

(ii) *for every $x_t < a_1\, n$, $\mathbb{P}(X_{t+1} > a_2\, n \mid X_t = x_t) = \exp\left(-n^{\Omega(1)}\right)$,*

(iii) $\mathbb{P}(|X_{t+1} - \mathbf{E}\left(X_{t+1} \mid X_t\right)| > n^{1/2+\varepsilon/4}) < 2\exp\left(-2n^{\varepsilon/2}\right)$,

then for $X_0 = \frac{a_2+a_3}{2} \cdot n$ and n large enough, we have w.h.p.

$$\inf\{t \in \mathbb{N}, X_t \geq a_3\, n\} \geq n^{1-\varepsilon}.$$

Proof. Let $T = n^{1-\epsilon}$. Let $Y_t := X_t - t$. We will consider the Doob decomposition of Y_t: for every $t \geq 1$, let

$$A_0 := 0 \quad \text{and for all } t > 0, \quad A_t := \sum_{k=1}^{t} \left[\mathbf{E}\left(Y_k \mid Y_{k-1}\right) - Y_{k-1}\right],$$

$$M_0 := Y_0 \quad \text{and for all } t > 0, \quad M_t := Y_0 + \sum_{k=1}^{t} \left[Y_k - \mathbf{E}\left(Y_k \mid Y_{k-1}\right)\right].$$

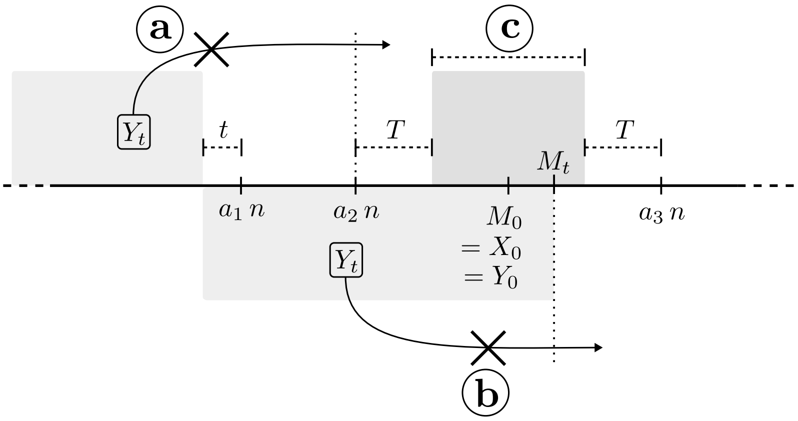


Figure 1 Sketch of the proof of Theorem 6. **(a)** By assumption (ii), with high probability, Y_t cannot jump from below $a_1\,n - t$ to above $a_2\,n - t$ in a single step, let alone $a_2\,n$. **(b)** In Claim 7, we use assumption (i) and the properties of the Doob's decomposition to show that, if $a_1\,n - t \le Y_t \le M_t$, then Y_t cannot jump above M_t in a single step. **(c)** In Claim 8, we use the Azuma-Hoeffding inequality to show that M_t remains in the interval $[a_2\,n + T, a_3\,n - T]$ for at least T rounds w.h.p. Overall, (a) (b) and (c) implies that Y_t must remain below $a_3\,n - T$ for at least T rounds w.h.p., yielding the desired conclusion.

With this definition, one can check that $Y_t = M_t + A_t$, and that $\{M_t\}_{t\in\mathbb{N}}$ is a martingale. The main ideas of the proof are depicted on Figure 1.

First, we show that by construction and Assumption (i), Y_t can never "jump over" M_t in one round, as long as it starts from the interval $\{a_1n - t, \dots, a_3n - t\}$.

▷ Claim 7. For every $t \in \mathbb{N}$,

$$M_t \ge Y_t \text{ and } Y_t \in \{a_1n - t, \dots, a_3n - t\} \implies M_{t+1} \ge Y_{t+1}.$$

Proof. Let $y_t \in \{a_1n - t, \dots, a_3n - t\}$ and $m_t \ge y_t$, and consider the event

$$\mathcal{E} := \left\{Y_t = y_t \bigcap M_t = m_t\right\}.$$

We have

$$\begin{aligned}\mathbf{E}\left(Y_{t+1} \mid Y_t = y_t\right) &= \mathbf{E}\left(X_{t+1} - (t+1) \mid X_t = y_t + t\right)\\ &= \mathbf{E}\left(X_{t+1} \mid X_t = y_t + t\right) - (t+1)\\ &\le (y_t + t + 1) - (t+1) && \text{(by (i))}\\ &= y_t.\end{aligned}$$

Therefore,

$$\left(A_{t+1} - A_t \mid \mathcal{E}\right) = \Big(\mathbf{E}\left(Y_{t+1} \mid Y_t\right) - Y_t \mid \mathcal{E}\Big) = \mathbf{E}\left(Y_{t+1} \mid Y_t = y_t\right) - y_t \le 0.$$

Since $m_t \ge y_t$, this implies

$$\left(A_{t+1} \mid \mathcal{E}\right) \le \left(A_t \mid \mathcal{E}\right) = \left(Y_t - M_t \mid \mathcal{E}\right) = y_t - m_t \le 0,$$

and thus,

$$\left(Y_{t+1} \mid \mathcal{E}\right) = \left(M_{t+1} + A_{t+1} \mid \mathcal{E}\right) \le \left(M_{t+1} \mid \mathcal{E}\right),$$

which concludes the proof of Claim 7. ◁

Now, using Azuma's inequality and Assumptions (ii) and (iii), we establish high probability bounds on the martingale M_t.

▷ Claim 8. With high probability, for every $t \leq T$, $a_2\,n + T < M_t < a_3\,n - T$.

Proof. By construction, and since X_t and Y_t only differ by a deterministic quantity,

$$M_{t+1} - M_t = Y_{t+1} - \mathbf{E}\left(Y_{t+1} \mid Y_t\right) = X_{t+1} - \mathbf{E}\left(X_{t+1} \mid X_t\right).$$

By assumption (iii) in the statement, this implies

$$\mathbb{P}\left(|M_{t+1} - M_t| > n^{\frac{1}{2}+\frac{\varepsilon}{4}}\right) \leq 2\exp\left(-2\,n^{\frac{\varepsilon}{2}}\right).$$

By the union bound,

$$\mathbb{P}\left(\exists s \leq t, |M_{s+1} - M_s| > n^{\frac{1}{2}+\frac{\varepsilon}{4}}\right) \leq 2\,t\exp\left(-2\,n^{\frac{\varepsilon}{2}}\right).$$

Let $\alpha = (a_3 - a_2)/4$, so that

$$M_0 + \alpha\,n = X_0 + \alpha\,n = \frac{a_2 + a_3}{2}\,n + \alpha\,n = a_3\,n - \alpha\,n, \quad \text{ and } M_0 - \alpha\,n = a_2\,n + \alpha\,n. \tag{6}$$

By the Azuma-Hoeffding inequality applied to $\{M_t\}_{t\in\mathbb{N}}$, we then have for every $t \leq T$,

$$\begin{aligned}
\mathbb{P}\left(|M_t - M_0| > \alpha n\right) &\leq 2\exp\left(-\frac{\alpha^2 n^2}{2\,t\,n^{1+\frac{\varepsilon}{2}}}\right) + 2\,t\exp\left(-2\,n^{\frac{\varepsilon}{2}}\right)\\
&\leq 2\exp\left(-\frac{\alpha^2 n^2}{2\,T\,n^{1+\frac{\varepsilon}{2}}}\right) + 2\,T\exp\left(-2\,n^{\frac{\varepsilon}{2}}\right) && \text{(since } t \leq T\text{)}\\
&= 2\exp\left(-\frac{\alpha^2}{2}\cdot n^{\frac{\varepsilon}{2}}\right) + 2\,T\exp\left(-2\,n^{\frac{\varepsilon}{2}}\right). && \text{(since } T = n^{1-\varepsilon}\text{)}
\end{aligned}$$

By the union bound,

$$\mathbb{P}\left(\exists t \leq T, |M_t - M_0| > \alpha n\right) \leq 2T\exp\left(-\frac{\alpha^2}{2}\cdot n^{\frac{\varepsilon}{2}}\right) + 2\,T^2\exp\left(-2\,n^{\frac{\varepsilon}{2}}\right) = o(n^{-2}). \tag{7}$$

Finally, note that for n large enough, $T = n^{1-\epsilon} < \alpha\,n$, and hence,

$$\begin{aligned}
&\mathbb{P}\left(\exists t \leq T, M_t \notin \{a_2\,n + T, \ldots, a_3\,n - T\}\right)\\
&\quad\leq \mathbb{P}\left(\exists t \leq T, M_t \notin \{a_2\,n + \alpha\,n, \ldots, a_3\,n - \alpha\,n\}\right)\\
&\quad= \mathbb{P}\left(\exists t \leq T, |M_t - M_0| > \alpha\,n\right) && \text{(by Equation (6))}\\
&\quad= o(n^{-2}), && \text{(by Equation (7))}
\end{aligned}$$

which concludes the proof of Claim 8. ◁

Next, we use Claim 7 and Claim 8 and Assumption (ii) of the theorem to show that Y_t can never jump over M_t, with high probability.

▷ Claim 9. With high probability, for every $t \leq T$, $M_t \geq Y_t$.

Proof. We will be conditioning on the two following events:

$$\begin{aligned}
\mathcal{E}_1 &:= \left\{\forall t \leq T, \quad Y_t \leq a_1\,n - t \implies Y_{t+1} \leq a_2\,n - t\right\},\\
\mathcal{E}_2 &:= \left\{\forall t \leq T, \quad a_2\,n + T < M_t < a_3\,n - T\right\}.
\end{aligned}$$

Note that $\mathcal{E}_2$ happen w.h.p. as a consequence of Claim 8. Moreover,

$$\begin{aligned}
\mathbb{P}(\mathcal{E}_1) &= 1 - \mathbb{P}\left(\bigcup_{t=0}^{T}\{Y_t \le a_1\,n - t\} \bigcap \{Y_{t+1} > a_2\,n - t\}\right)\\
&= 1 - \mathbb{P}\left(\bigcup_{t=0}^{T}\{X_t \le a_1\,n\} \bigcap \{X_{t+1} > a_2\,n\}\right)\\
&\ge 1 - \sum_{t=0}^{T}\mathbb{P}\left(X_t \le a_1\,n \;\bigcap\; X_{t+1} > a_2\,n\right) && \text{(by the union bound)}\\
&\ge 1 - \sum_{t=0}^{T}\mathbb{P}\left(X_{t+1} > a_2\,n \mid X_t \le a_1\,n\right)\\
&\ge 1 - T\exp\left(-n^{\Omega(1)}\right) && \text{(by assumption (ii))}\\
&\ge 1 - o(n^{-2}). && \text{(since } T = n^{1-\varepsilon}\text{)}
\end{aligned}$$

Hence, $\mathcal{E}_1$ also happens w.h.p., and so does $\mathcal{E}_1 \bigcap \mathcal{E}_2$. To conclude the proof, we will show that

$$\mathcal{E}_1 \bigcap \mathcal{E}_2 \implies \forall t \le T, M_t \ge Y_t. \tag{8}$$

We will proceed by induction on t. By definition, we have $M_0 = Y_0$. Now, let $t < T$, and consider the case that $\mathcal{E}_1$ and $\mathcal{E}_2$ hold, and that $M_t \ge Y_t$.

- If $Y_t < a_1\,n - t$, we have

$$Y_{t+1} \underset{(\mathcal{E}_1)}{\le} a_2\,n - t \le a_2\,n + T \underset{(\mathcal{E}_2)}{<} M_{t+1}.$$

- Otherwise, by induction hypothesis and $\mathcal{E}_2$, we have $a_1\,n - t \le Y_t \le M_t < a_3 n - T < a_3 n - t$. In this case, $M_{t+1} \ge Y_{t+1}$ follows as a consequence of Claim 7.

By induction, we deduce that Equation (8) holds, which concludes the proof of Claim 9. ◁

Finally, we are ready to conclude. By Claim 9, with high probability, for every $t \le T$, $X_t \le M_t + t$. Therefore,

$$\inf\{t \in \mathbb{N}, X_t > a_3 n\} \ge \inf\{t \in \mathbb{N}, M_t > a_3 n - t\} \quad \text{w.h.p.}$$

Moreover, by Claim 8,

$$\inf\{t \in \mathbb{N}, M_t > a_3 n - t\} > T \quad \text{w.h.p.},$$

which gives the desired result. ◁

By symmetry, the following result can be deduced directly from Theorem 6.

▶ **Corollary 10.** *Let $\{X_t\}$ be a Markov chain on $\mathbb{Z}$ and $\varepsilon > 0$. If there are $a_1 < a_2 < a_3 \in \mathbb{R}$ s.t.*

(i) *for every $x_t \in \{\lceil a_1\,n\rceil, ..., \lfloor a_3\,n\rfloor\}$, $\mathbf{E}(X_{t+1} \mid X_t = x_t) \ge x_t - 1$,*
(ii) *for every $x_t > a_3\,n$, $\mathbb{P}(X_{t+1} < a_2\,n \mid X_t = x_t) = \exp(-n^{\Omega(1)})$,*
(iii) *$\mathbb{P}(|X_{t+1} - \mathbf{E}\left(X_{t+1} \mid X_t\right)| > n^{1/2+\varepsilon/4}) < 2\exp\left(-2n^{\varepsilon/2}\right)$,*
then for $X_0 = \frac{a_1+a_2}{2} \cdot n$ and for n large enough, we have w.h.p.

$$\inf\{t \in \mathbb{N}, X_t \le a_1\,n\} \ge n^{1-\varepsilon}.$$

Proof. It is easy to check that if the assumptions of Corollary 10 hold for $\{X_t\}_{t\in\mathbb{N}}$ and some constants $a_1 < a_2 < a_3$, then the assumptions of Theorem 6 hold w.r.t. $\{-X_t\}_{t\in\mathbb{N}}$ and $-a_3 < -a_2 < -a_1$ respectively – and the conclusion follows. ◀

4 The Main Proof

4.1 The Voter Dynamics

First, we focus on the Voter dynamics (Algorithm 1) and show that it satisfies the lower bound stated in Theorem 1, which we will prove in its full generality in Section 4.2. We start by observing that for g^{voter} defined in Equation (13), and by definition in Equation (1),

$$F_n^{\text{voter}}(p) = -p + \sum_{k=0}^{\ell} \binom{\ell}{k} p^k (1-p)^{\ell-k} \cdot \frac{k}{\ell} = -p + p = 0.$$

Then, we conclude by applying the following result. Due to space constraints, its proof is deferred to Appendix D.

▶ **Lemma 11.** *Consider a protocol $\mathcal{P}$ satisfying $F_n = 0$ for every n large enough. There exists a sequence of configurations $\{C_n\}$ such that for every $\epsilon > 0$, with high probability,*

$$\tau_n(\mathcal{P}, C_n) > n^{1-\varepsilon}.$$

4.2 General Case

Our main result (Theorem 1) will follow as a consequence of Theorem 12 below.

▶ **Theorem 12.** *For every $\varepsilon > 0$ and every protocol $\mathcal{P}$, there exists an infinite set $\mathcal{S} \subseteq \mathbb{N}$ and a sequence of configurations $\{C_n\}$ such that for every $n \in \mathcal{S}$,*

$$\mathbb{P}\left(\tau_n(\mathcal{P}, C_n) > n^{1-\varepsilon}\right) \geq 1 - \frac{1}{n^{\Omega(1)}}.$$

In other words, the convergence time of $\mathcal{P}$ restricted to $\mathcal{S}$ is greater than $n^{1-\varepsilon}$ w.h.p.

Proof. Recall the definition of F_n in Equation (1). If there is $N \in \mathbb{N}$ s.t. for every $n \geq N$, $F_n = 0$, then we can conclude by applying Lemma 11. Otherwise, there is an infinite set $\mathcal{S}_0 \subseteq \mathbb{N}$ s.t. for every $n \in \mathcal{S}_0$, $F_n \neq 0$, which we will assume from now on.

By definition in Equation (1), F_n is a polynomial of degree at most $\ell + 1$. For $n \in \mathcal{S}_0$, let d_n be the number of roots of F_n in the interval $[0,1]$ (counted with multiplicity). By Proposition 3, $g_n^{[0]}(0) = 0$ and $g_n^{[1]}(\ell) = 1$, so $F_n(0) = F_n(1) = 0$, and thus $d_n \in \{2, \dots, \ell+1\}$. Since d_n can only adopt finitely many values, there exists $d \in \{2, \dots, \ell+1\}$ and an infinite set $\mathcal{S}_1 \subseteq \mathcal{S}_0$ s.t. for every $n \in \mathcal{S}_1$, $d_n = d$.

For $n \in \mathcal{S}_1$, let $0 = r_n^{(1)} \leq \ldots \leq r_n^{(d)} = 1$ be the roots of F_n within the interval $[0,1]$, with multiplicity, in increasing order. The sequence $\left\{(r_n^{(1)}, \dots, r_n^{(d)})\right\}_{n \in \mathcal{S}_1}$ is bounded in $\mathbb{R}^d$ by definition. Hence, by the Bolzano-Weierstrass theorem, there exists a converging sub-sequence of $\left\{(r_n^{(1)}, \dots, r_n^{(d)})\right\}_{n \in \mathcal{S}_1}$; i.e., there are $0 = r_\infty^{(1)} \leq \ldots \leq r_\infty^{(d)} = 1$ together with an infinite set $\mathcal{S}_2 \subseteq \mathcal{S}_1$ s.t. for every $k \in [d]$,

$$\lim_{\substack{n \to +\infty \\ n \in \mathcal{S}_2}} r_n^{(k)} = r_\infty^{(k)}. \tag{9}$$

Let $k_0 := \min\{k \in [d], r_\infty^{(k)} = 1\}$. Note that $k_0 \geq 2$ (since $r_\infty^{(1)} = 0$) and by definition, $r_\infty^{(k_0-1)} < 1 = r_\infty^{(k_0)}$. Moreover, for every $n \in \mathcal{S}_2$, F_n is non-zero and has constant sign on $(r_n^{(k_0-1)}, r_n^{(k_0)})$. Therefore, there exists an infinite set $\mathcal{S}_3 \subseteq \mathcal{S}_2$ s.t.

1. either $\forall n \in \mathcal{S}_3$, $F_n < 0$ on $(r_n^{(k_0-1)}, r_n^{(k_0)})$,
2. or $\forall n \in \mathcal{S}_3$, $F_n > 0$ on $(r_n^{(k_0-1)}, r_n^{(k_0)})$.

In the remainder of the proof, we will analyse these two cases separately. The reader is strongly encouraged to consult Figure 2 and 3 respectively for a clearer understanding of the argument.

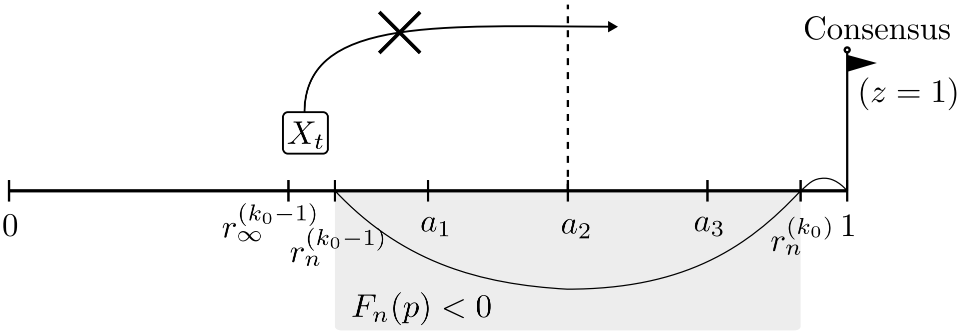


Figure 2 Illustration of the arguments for Case 1. We consider a configuration in which the correct opinion is 1. Constant a_1 is fixed arbitrarily in the interval $(r_\infty^{(k_0-1)}, 1)$ Then, a_2 is chosen according to Proposition 4 to ensure that X_t cannot jump from below $a_1\,n$ to above $a_2\,n$. Finally, a_3 is set anywhere in the interval $(a_2, 1]$. By assumption, $F_n < 0$ on $[a_1, a_3]$, and we can eventually apply Theorem 6.

Case 1

Let $a_1 \in (r_\infty^{(k_0-1)}, 1)$. Let $a_2 = a_2(a_1, \ell) \in (a_1, 1)$ given by Proposition 4, s.t. for n large enough,

$$\text{for every } x_t \le a_1\,n, \quad \mathbb{P}(X_{t+1} \le a_2\,n \mid X_t = x_t) \ge 1 - \exp(-2\,n^{-1/2}).$$

Let $a_3 \in (a_2, 1)$. By Equation (9), for n large enough, $r_n^{(k_0-1)} < a_1$ and $r_n^{(k_0)} > a_3$. We now wish to use Theorem 6, with a_1, a_2, a_3 as we just defined. Let us check that every assumption holds:

- For every $x_t \in \{a_1\,n, \dots, a_3\,n\}$, we have $x_t/n \in [a_1, a_3] \subset [r_n^{(k_0-1)}, r_n^{(k_0)}]$, and so $F_n(x_t/n) < 0$ by assumption. Therefore, Equation (3) gives

$$\mathbf{E}\left(X_{t+1} \mid X_t = x_t\right) \le x_t + n\,F_n\left(\frac{x_t}{n}\right) + 1 < x_t + 1, \tag{10}$$

  so assumption (i) in the statement of Theorem 6 holds.
- Assumption (ii) holds by Proposition 4.
- Finally, assumption (iii) follows from Hoeffding's bound: conditioning on X_t, X_{t+1} is the sum of n Bernoulli random variables, then the result follow choosing $\delta = n^{1/2+\varepsilon/4}$.

Assuming that the source has opinion $z = 1$, we apply Theorem 6, which implies the existence of an initial configuration C_n s.t. the convergence time is bounded w.h.p.:

$$\tau_n(g, C_n) = \inf\{t \in \mathbb{N}, X_t = n\} \ge \inf\{t \in \mathbb{N}, X_t \ge a_3\,n\} \ge n^{1-\varepsilon}.$$

Case 2

Let $a_1, a_2, a_3 \in (r_\infty^{(k_0-1)}, 1)$, with $r_\infty^{(k_0-1)} < a_1 < a_2 < a_3 < 1$. First, we show that by taking n large enough, we can have F_n be arbitrarily close to 0 on the interval $[r_n^{(k_0)}, 1]$.

▷ Claim 13. For every $\delta > 0$, for n large enough: $r_n^{(k_0)} > (1+a_3)/2$, and for every $p \in [r_n^{(k_0)}, 1]$, $F_n(p) > -\delta$.

Proof. Let $\delta > 0$ and $n \in \mathcal{S}_2$. Since F_n has bounded coefficients and degree d, and since $F_n(r_n^{(k_0)}) = F_n(1) = 0$, by Claim 17 in Appendix D we obtain the existence of C_0 s.t.

$$\text{for every } p \in [r_n^{(k_0)}, 1], \quad |F_n(p)| < C_0 \cdot (1 - r_n^{(k_0)}). \tag{11}$$

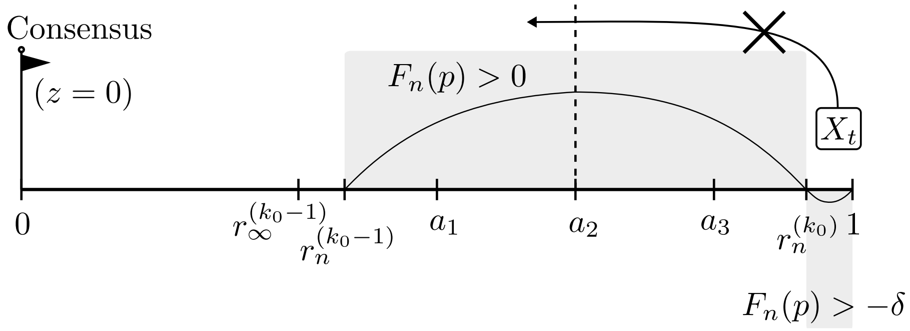


Figure 3 Illustration of the arguments for Case 2. We consider a configuration in which the correct opinion is 0. Constants a_1, a_2 and a_3 are chosen arbitrarily in the interval $(r_\infty^{(k_0-1)}, 1)$. By assumption, $F_n > 0$ on $[a_1, a_3]$. Moreover, once a_2 and a_3 are fixed, we give a lower-bound on F_n on the interval $[r_n^{(k_0)}, 1]$, by letting $r_n^{(k_0)}$ be sufficiently close to 1, in order to ensure that X_t cannot jump from above $a_3\, n$ to below $a_2\, n$. Eventually, we are able to apply Corollary 10.

By Equation (9) and by definition of k_0, $r_n^{(k_0)}$ tends to 1 as n goes to $+\infty$. If n is large enough,

$$r_n^{(k_0)} > \max\left\{1 - \frac{\delta}{C_0}, \frac{1+a_3}{2}\right\}. \tag{12}$$

By Equation (11), this implies that $|F_n(p)| < \delta$ on $[r_n^{(k_0)}, 1]$, which, together with Equation (12), concludes the proof of Claim 13. ◁

Now, we use the previous result to establish a lower bound on $p + F_n(p)$ when $p \geq a_1$.

▷ Claim 14. For n large enough,

- for every $p \in [a_1, a_3]$, $p + F_n(p) > p$.
- for every $p \in [a_3, 1]$, $p + F_n(p) > a_3$.

Proof. For $p \in [a_1, r_n^{(k_0)})$, $F_n(p) > 0$ by assumption. Therefore:

- for every $p \in [a_1, a_3]$, $p + F_n(p) > p$.
- for every $p \in [a_3, r_n^{(k_0)})$, $p + F_n(p) > p \geq a_3$.

All is left to prove is that $p + F_n(p) > a_3$ on $[r_n^{(k_0)}, 1]$. Let $\delta = (1 - a_3)/2$, and let n be large enough for Claim 13 to hold w.r.t. δ. For every $p \in [r_n^{(k_0)}, 1]$, we have

$$\begin{aligned} p + F_n(p) &> r_n^{(k_0)} - \delta && \text{(by Claim 13 and definition of } p\text{)} \\ &= a_3 + \left(\frac{1-a_3}{2} - \delta\right) + \left(r_n^{(k_0)} - \frac{1+a_3}{2}\right) \\ &> a_3, && \text{(by Claim 13 and definition of } \delta\text{)} \end{aligned}$$

which concludes the proof of Claim 14. ◁

Finally, similarly to the first case, we use Corollary 10 to conclude. Again, we start by checking that all assumptions hold. Let n be large enough for Claim 14 to hold.

- For every $x_t \in \{a_1\, n, \ldots, a_3\, n\}$, we have $x_t/n + F_n(x_t/n) > x_t/n$ by Claim 14. Therefore, Equation (4) rewrites

$$\mathbf{E}\left(X_{t+1} \mid X_t = x_t\right) \geq x_t + n\, F_n\left(\frac{x_t}{n}\right) - 1 \geq x_t - 1,$$

so assumption (i) in the statement of Corollary 10 holds.

- For every $x_t \in \{a_3 n, \dots, n\}$, we have $x_t/n + F_n(x_t/n) > a_3$ by Claim 14. Therefore, Equation (4) rewrites

$$\mathbf{E}\left(X_{t+1} \mid X_t = x_t\right) \geq x_t + n\, F_n\left(\frac{x_t}{n}\right) - 1 \geq a_3\, n - 1.$$

Therefore, by Hoeffding's bound,

$$\begin{aligned}\mathbb{P}\left(X_{t+1} < a_2\, n \mid X_t = x_t\right) &\geq \mathbb{P}\left(X_{t+1} < \mathbf{E}\left(X_{t+1}\right) - \frac{a_3 - a_2}{2} n \mid X_t = x_t\right) \\ &\leq \exp\left(-2\left(\frac{a_3 - a_2}{2}\right)^2 n\right),\end{aligned}$$

and so assumption (ii) holds.

- Finally, assumption (iii) follows from Hoeffding's bound, as in the first case.

Assuming that the source has opinion $z = 0$, we apply Corollary 10, which implies the existence of an initial configuration C_n s.t. the convergence time is bounded:

$$\tau_n(g, C_n) = \inf\{t \in \mathbb{N}, X_t = 0\} \geq \inf\{t \in \mathbb{N}, X_t \leq a_1 n\} \geq n^{1-\varepsilon}.$$ ◀

5 Discussion and Future Works

In this paper, we explore the minimal requirements for simultaneously reaching consensus and propagating information in a distributed system. We consider memory-less and anonymous agents, which update their opinion synchronously after observing the opinions of a few other agents sampled uniformly at random, and whose goal is to converge on the correct opinion held by a single "source" individual. In addition, we adopt the self-stabilizing framework, which in a memory-less setting, means that convergence must happen for any possible initialization of the opinions of the agents (including the source). Under this model, we show that to obtain a convergence time better than $n^{1-\epsilon}$, the number ℓ of samples obtained by each agent in every round must necessarily tend towards infinity as n increases. Our result extends the range of values of ℓ for which the performance of the "minority" dynamics (Algorithm 2) is characterized. Our technique, which consists in translating the sample size into the degree of a well-chosen polynomial, and then inspecting its roots, is simple yet quite novel (to the best of our knowledge), and may be used to show similar results in other settings.

Our ultimate goal is to fully characterize the complexity of the bit-dissemination problem in the parallel setting and in the absence of memory, as a function of the sample size. Regarding values of ℓ allowing poly-logarithmic convergence time, there is still a large gap between our lower bound $\ell = \Omega(1)$ and the upper bound $\ell = O(\sqrt{n \log n})$ mentioned in [7]. Closing or narrowing this gap, even specifically for the minority dynamics, would be of appreciable interest in our opinion.

Another natural continuation would be to generalize our result to protocols using a constant amount of memory. If feasible, the resulting lower bound would still be compatible with the algorithm of [23], which requires $\Omega(\log\log n)$ bits of memory.

References

1 Yehuda Afek, Noga Alon, Omer Barad, Eran Hornstein, Naama Barkai, and Ziv Bar-Joseph. A biological solution to a fundamental distributed computing problem. *Science*, 331(6014):183–185, January 2011. doi:10.1126/science.1193210.

2 James Aspnes and Eric Ruppert. An introduction to population protocols. In Benoît Garbinato, Hugo Miranda, and Luís Rodrigues, editors, *Middleware for Network Eccentric and Mobile Applications*, pages 97–120. Springer, Berlin, Heidelberg, 2009. doi:10.1007/978-3-540-89707-1_5.

3 M. Ballerini, N. Cabibbo, R. Candelier, A. Cavagna, E. Cisbani, I. Giardina, V. Lecomte, A. Orlandi, G. Parisi, A. Procaccini, M. Viale, and V. Zdravkovic. Interaction ruling animal collective behavior depends on topological rather than metric distance: Evidence from a field study. *Proceedings of the National Academy of Sciences of the United States of America*, 105(4):1232–1237, January 2008. doi:10.1073/pnas.0711437105.

4 Paul Bastide, George Giakkoupis, and Hayk Saribekyan. Self-stabilizing clock synchronization with 1-bit messages. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Proceedings, pages 2154–2173. Society for Industrial and Applied Mathematics, January 2021. doi:10.1137/1.9781611976465.129.

5 Luca Becchetti, Andrea Clementi, Amos Korman, Francesco Pasquale, Luca Trevisan, and Robin Vacus. On the role of memory in robust opinion dynamics. In Edith Elkind, editor, *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, volume 1, pages 29–37, Macao, August 2023. International Joint Conferences on Artificial Intelligence Organization. doi:10.24963/ijcai.2023/4.

6 Luca Becchetti, Andrea Clementi, and Emanuele Natale. Consensus dynamics: An overview. *ACM SIGACT News*, 51(1):58–104, March 2020. doi:10.1145/3388392.3388403.

7 Luca Becchetti, Andrea Clementi, Francesco Pasquale, Luca Trevisan, Robin Vacus, and Isabella Ziccardi. The minority dynamics and the power of synchronicity. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Proceedings, pages 4155–4176. Society for Industrial and Applied Mathematics, Alexandria, Virginia, U.S., January 2024. doi:10.1137/1.9781611977912.144.

8 William Bialek, Andrea Cavagna, Irene Giardina, Thierry Mora, Edmondo Silvestri, Massimiliano Viale, and Aleksandra M. Walczak. Statistical mechanics for natural flocks of birds. *Proceedings of the National Academy of Sciences*, 109(13):4786–4791, March 2012. doi:10.1073/pnas.1118633109.

9 Lucas Boczkowski, Amos Korman, and Emanuele Natale. Minimizing message size in stochastic communication patterns: Fast self-stabilizing protocols with 3 bits. In *Proceedings of the 2017 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Proceedings, pages 2540–2559. Society for Industrial and Applied Mathematics, January 2017. doi:10.1137/1.9781611974782.168.

10 Fan Chung and Linyuan Lu. Concentration inequalities and martingale inequalities: A survey. *Internet Mathematics*, 3(1):79–127, January 2006. doi:10.1080/15427951.2006.10129115.

11 Andrea Clementi, Francesco d'Amore, George Giakkoupis, and Emanuele Natale. Search via parallel Lévy walks on z2. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 81–91, New York, NY, USA, July 2021. Association for Computing Machinery. doi:10.1145/3465084.3467921.

12 Emilio Cruciani, Hlafo Alfie Mimun, Matteo Quattropani, and Sara Rizzo. Phase transitions of the k-majority dynamics in a biased communication model. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking*, ICDCN '21, pages 146–155, New York, NY, USA, January 2021. Association for Computing Machinery. doi:10.1145/3427796.3427811.

13 Emilio Cruciani, Emanuele Natale, André Nusser, and Giacomo Scornavacca. Phase transition of the 2-Choices dynamics on core–periphery networks. *Distributed Computing*, 34(3):207–225, June 2021. doi:10.1007/s00446-021-00396-5.

14 Étienne Danchin, Luc-Alain Giraldeau, Thomas J. Valone, and Richard H. Wagner. Public information: From nosy neighbors to cultural evolution. *Science*, 305(5683):487–491, July 2004. doi:10.1126/science.1098254.

15 Bartłomiej Dudek and Adrian Kosowski. Universal protocols for information dissemination using emergent signals. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 87–99, New York, NY, USA, June 2018. Association for Computing Machinery. doi:10.1145/3188745.3188818.

16 Yuval Emek and Roger Wattenhofer. Stone age distributed computing. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 137–146, New York, NY, USA, July 2013. Association for Computing Machinery. doi:10.1145/2484239.2484244.

17 Ofer Feinerman and Amos Korman. Individual versus collective cognition in social insects. *Journal of Experimental Biology*, 220(1):73–82, January 2017. doi:10.1242/jeb.143891.

18 Ofer Feinerman, Amos Korman, Zvi Lotker, and Jean-Sebastien Sereni. Collaborative search on the plane without communication. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 77–86, New York, NY, USA, July 2012. Association for Computing Machinery. doi:10.1145/2332432.2332444.

19 Mikaela Irene D. Fudolig and Jose Perico H. Esguerra. Analytic treatment of consensus achievement in the single-type zealotry voter model. *Physica A: Statistical Mechanics and its Applications*, 413:626–634, November 2014. doi:10.1016/j.physa.2014.07.033.

20 Mohsen Ghaffari, Cameron Musco, Tsvetomira Radeva, and Nancy Lynch. Distributed house-hunting in ant colonies. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 57–66, New York, NY, USA, July 2015. Association for Computing Machinery. doi:10.1145/2767386.2767426.

21 Yehuda Hassin and David Peleg. Distributed probabilistic polling and applications to proportionate agreement. *Information and Computation*, 171(2):248–268, December 2001. doi:10.1006/inco.2001.3088.

22 Yael Katz, Kolbjørn Tunstrøm, Christos C. Ioannou, Cristián Huepe, and Iain D. Couzin. Inferring the structure and dynamics of interactions in schooling fish. *Proceedings of the National Academy of Sciences*, 108(46):18720–18725, November 2011. doi:10.1073/pnas.1107583108.

23 Amos Korman and Robin Vacus. Early adapting to trends: Self-stabilizing information spread using passive communication. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, pages 235–245, New York, NY, USA, July 2022. Association for Computing Machinery. doi:10.1145/3519270.3538415.

24 Melissa B. Miller and Bonnie L. Bassler. Quorum sensing in bacteria. *Annual Review of Microbiology*, 55(1):165–199, 2001. doi:10.1146/annurev.micro.55.1.165.

25 V. Moeinifar and S. Gündüç. Zealots' effect on opinion dynamics in complex networks. *Mathematical Modeling and Computing*, 8(2):pp. 203, Wed, 05/05/2021 - 17:53.

26 Arpan Mukhopadhyay, Ravi R. Mazumdar, and Rahul Roy. Binary opinion dynamics with biased agents and agents with different degrees of stubbornness. In *2016 28th International Teletraffic Congress (ITC 28)*, pages 261–269, Würzburg, Germany, September 2016. IEEE. doi:10.1109/ITC-28.2016.143.

27 V. Sood and S. Redner. Voter model on heterogeneous graphs. *Physical Review Letters*, 94(17):178701, May 2005. doi:10.1103/PhysRevLett.94.178701.

28 David J. T. Sumpter, Jens Krause, Richard James, Iain D. Couzin, and Ashley J. W. Ward. Consensus decision making by fish. *Current Biology*, 18(22):1773–1777, November 2008. doi:10.1016/j.cub.2008.09.064.

29 Anthony Trewavas. Plant intelligence: Mindless mastery. *Nature*, 415(6874):841–841, February 2002. doi:10.1038/415841a.

30 Tamás Vicsek and Anna Zafeiris. Collective motion. *Physics Reports*, 517(3):71–140, August 2012. doi:10.1016/j.physrep.2012.03.004.

31 Ercan Yildiz, Asuman Ozdaglar, Daron Acemoglu, Amin Saberi, and Anna Scaglione. Binary opinion dynamics with stubborn agents. *ACM Transactions on Economics and Computation*, 1(4):19:1–19:30, December 2013. doi:10.1145/2538508.

A Well-known Dynamics

In this section, we define two important dynamics. Algorithms 1 and 2 below only describe the behaviour of non-source agents, since the source never changes its opinion.

Algorithm 1 Voter dynamics.

```
Input: An opinion sample $S$ of
       size $\ell$.
$X_{t+1}^{(i)} \leftarrow$ a random opinion in $S$
(Since $S$ is already sampled
  uniformly at random, this protocol
  yields the same dynamics
  regardless of the value of $\ell$, and is
  usually defined with $\ell = 1$).
```

Algorithm 2 Minority dynamics [7].

```
Input: An opinion sample $S$ of
       size $\ell$.
if all opinions in $S$ are equal to $x$
 then
    $X_{t+1}^{(i)} \leftarrow x$
else
    $X_{t+1}^{(i)} \leftarrow$ minority opinion in $S$
    (breaking ties randomly).
```

In terms of our definition, the Voter dynamics writes

$$g_n^{[0]}(k) = g_n^{[1]}(k) =: g^{\text{voter}}(k) = \frac{k}{\ell}, \qquad \text{for every } k \in \{0, \dots, \ell\}. \tag{13}$$

Similarly, given that ties are broken u.a.r., the Minority dynamics is given by

$$g_n^{[0]}(k) = g_n^{[1]}(k) =: g^{\text{minority}}(k) = \begin{cases} 1 & \text{if } k = \ell \text{ or } 0 < k < \frac{\ell}{2}, \\ \frac{1}{2} & \text{if } k = \frac{\ell}{2}, \\ 0 & \text{if } k = 0 \text{ or } \frac{\ell}{2} < k < \ell. \end{cases} \tag{14}$$

B Probabilistic Tools

▶ **Theorem 15** (Hoeffding's bound). *Let $X_1, \dots, X_n$ be i.i.d. random variables taking values in $\{0,1\}$, let $X = \sum_{i=1}^n X_i$ and $\mu = \mathbf{E}(X) = n\mathbb{P}(X_1 = 1)$. Then it holds for all $\delta > 0$ that*

$$\mathbb{P}\left(X \leq \mu - \delta\right), \mathbb{P}\left(X \geq \mu + \delta\right) \leq \exp\left(-\frac{2\delta^2}{n}\right).$$

The following version of Azuma's inequality, which accounts for the possibility that the martingale makes a large jump with a small probability, appears in [10, Section 8, p.34].

▶ **Theorem 16** (Azuma-Hoeffding inequality). *Let $(X_t)_{t\in\mathbb{N}}$ be a martingale, and let $T \in \mathbb{N}$. If there is $p > 0$ and $c_1, \dots, c_T$ such that*

$$\mathbb{P}\left(\exists t \leq T, X_t - X_{t-1} > c_t\right) \leq p,$$

then for every $\delta > 0$,

$$\mathbb{P}\left(|X_T - X_0| > \delta\right) \leq 2\exp\left(-\frac{\delta^2}{2\sum_{t=1}^T c_t^2}\right) + p.$$

C Upper Bound for the Voter Dynamics

The following proof involves only classical arguments (see, for instance, [31] and [21, Section 2.4]). We nonetheless present it here for the sake of completeness.

Proof of Theorem 2. The reader who is not already familiar with the idea is strongly encouraged to refer to Figure 4 for an illustration.

Without loss of generality, we consider the Voter dynamics with $\ell = 1$. In this case, the sample $S_t^{(i)}$ of a non-source agent $i \neq 1$ in round t is simply an element of the set $I = \{1, \ldots, n\}$ of all agents, drawn uniformly at random. In the case of the source, for the sake of the argument, we let $S_t^{(1)} = 1$ for every $t \in \mathbb{N}$, i.e., we consider that the source agent applies the Voter rule but always samples itself.

Now, let us fix an horizon T. We will proceed by examining n random walks $\{W_{T-t}^{(i)}\}_{t \leq T}$, defined on the same randomness, but for which the time flows backward. Specifically, for every $i \in I$, let $W_T^{(i)} = i$, i.e., every random walk "starts" at a different position. Moreover, for $t < T$, let

$$W_t^{(i)} := S_t^{\left(W_{t+1}^{(i)}\right)}.$$

In other words, if a random walk is in position j in round $t+1$, and if Agent j samples i in round t, then the random walk "moves" to i in round t. Note that by definition, if a random walk moves to position 1 in round t, it will remain in position 1 for all remaining rounds:

$$W_t^{(i)} = 1 \implies \forall s \in \{1, \ldots, t\}, W_s^{(i)} = 1. \tag{15}$$

Moreover, if the random walk indexed by i ends up in position 1 (in round 1), it implies that the agent of index $W_t^{(i)}$ holds the correct opinion in round t:

$$W_1^{(i)} = 1 \implies X_t^{\left(W_t^{(i)}\right)} = z. \tag{16}$$

To show that, we can simply proceed by induction on t, using the definition of the random walks. Equation (15) and Equation (16) together implies that if the random walks indexed by i ever moves to position 1, then Agent i has the correct opinion in round T:

$$\exists t \leq T, W_t^{(i)} = 1 \implies X_T^{(i)} = z. \tag{17}$$

Therefore, for every $i \neq 1$,

$$\begin{aligned}
\mathbb{P}\left(X_T^{(i)} \neq z\right) &\leq \mathbb{P}\left(\forall t \leq T, W_t^{(i)} \neq 1\right) && \text{(by Equation (17))}\\
&\leq \prod_{t=0}^{T-1} \mathbb{P}\left(S_t^{\left(W_{t+1}^{(i)}\right)} \neq 1 \mid W_{t+1}^{(i)} \neq 1\right) && \text{(by the chain rule)}\\
&= \left(1 - \frac{1}{n}\right)^T. && \text{(uniform and independent samples)}
\end{aligned}$$

Note that this bound holds trivially for $i = 1$. Therefore, by the union bound,

$$\mathbb{P}\left(\forall i \in I, X_T^{(i)} = z\right) = 1 - \mathbb{P}\left(\exists i \in I, X_T^{(i)} \neq z\right) \geq 1 - \sum_{i \in I} \mathbb{P}\left(X_T^{(i)} \neq z\right) \geq 1 - n\left(1 - \frac{1}{n}\right)^T.$$

Taking $T = 2n \log n$, for n large enough,

$$\left(1 - \frac{1}{n}\right)^{2n\log n} = \exp\left(2n \log n \log\left(1 - 1/n\right)\right) \leq \exp\left(-2\log n\right) = \frac{1}{n^2},$$

which concludes the proof of Theorem 2. ◀

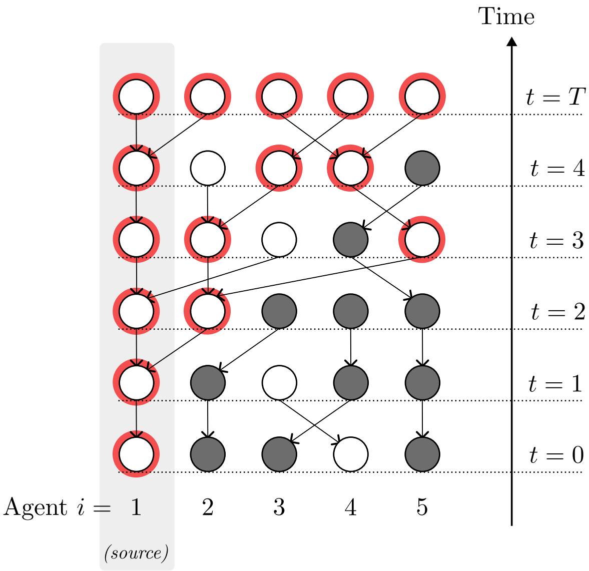


Figure 4 Depiction of the dual process behind the proof of Theorem 2. The color of the circle in row t, column i, corresponds to the opinion of Agent i in round t: it is black if $X_t^{(i)} = 1$, and white otherwise. An arrow is drawn from $(i, t+1)$ to (j, t) if $S_t^{(i)} = j$, i.e., if Agent i observes Agent j in round t (and thus adopts their opinion in round $t + 1$). Red circles depict the locations of n coalescing random walks going backward in time, and initially present at every location. Random walks at a location $i > 1$ make a move using the same randomness as the samples, while the source acts like a sink. If all random walks have coalesced in less than T rounds, it implies that the opinion of each agent in round T comes from the source, and thus that the dynamics has reached consensus on the correct opinion.

D Missing Proofs

▷ Claim 17. For every M, d, there exists $C_0 = C_0(M, d) > 0$ s.t. for every polynomial P of degree d and coefficients bounded by M, every $a, b \in [0, 1]$ with $P(a) = P(b) = 0$, and every $x \in [a, b]$, $P(x) < C_0 \cdot (b - a)$.

Proof. Since P has degree d and coefficients bounded by M, there exists $C = C(M, d)$ s.t. $|P'(x)| < C$ on $[0, 1]$. Therefore, for every $x \in [a, (a + b)/2]$, we have

$$|P(x)| = |P(x) - P(a)| < C \cdot (x - a) < C \cdot \frac{b - a}{2}.$$

Similarly, for every $x \in [(a + b)/2, b]$, we have

$$|P(x)| = |P(b) - P(x)| < C \cdot (b - x) < C \cdot \frac{b - a}{2}.$$

Taking $C_0 = C/2$ concludes the proof of Claim 17. ◁

Proof of Lemma 11. Let $\varepsilon > 0$. We want to apply Theorem 6 with $a_1 = 1/4$, $a_2 = 1/2$, $a_3 = 3/4$. If the three hypotheses hold, we have for the initial configuration $C_n := (z = 1, X_0 = \frac{a_2\, n + a_3\, n}{2})$,

$$\tau_n(\mathcal{P}, C_n) \geq \inf\{t \in \mathbb{N}, X_t > a_3 n\} \geq n^{1-\varepsilon}.$$

Now, let us show that the hypotheses hold:

- **Proving (i).** Since $F_n = 0$ for n large enough, we have by Proposition 5 that $\mathbf{E}(X_{t+1} \mid X_t = x_t) \leq x_t + 1$.
- **Proving (iii).** Conditioning on X_t, X_{t+1} is a sum of n independent Bernoulli random variables. Therefore, we can use Hoeffding's bound to obtain

  $$\mathbb{P}(|X_{t+1} - \mathbf{E}\left(X_{t+1} \mid X_t\right)| > n^{1/2+\varepsilon/4}) < 2\exp\left(-2n^{\varepsilon/2}\right),$$

  which establishes (iii).
- **Proving (ii).** If $x_t < a_1 n$ and n is large enough, (ii) follows again from Hoeffding's bound:

  $$\begin{aligned}\mathbb{P}(X_{t+1} > a_2 n \mid X_t = x_t) &\leq \mathbb{P}(X_{t+1} > n^{1/2+1/4} + \mathbf{E}\left(X_{t+1} \mid X_t = x_t\right) \mid X_t = x_t) \\ &< 2\exp\left(-2n^{1/2}\right),\end{aligned}$$

which establishes (ii) and concludes the proof of Lemma 11. ◀

Parallel Set Cover and Hypergraph Matching via Uniform Random Sampling

Laxman Dhulipala
Google Research, New York, NY, USA

Michael Dinitz
Johns Hopkins University, Baltimore, MD, USA

Jakub Łącki
Google Research, New York, NY, USA

Slobodan Mitrović
UC Davis, CA, USA

Abstract

The SetCover problem has been extensively studied in many different models of computation, including parallel and distributed settings. From an approximation point of view, there are two standard guarantees: an $O(\log \Delta)$-approximation (where Δ is the maximum set size) and an $O(f)$-approximation (where f is the maximum number of sets containing any given element).

In this paper, we introduce a new, surprisingly simple, model-independent approach to solving SetCover in unweighted graphs. We obtain multiple improved algorithms in the MPC and CRCW PRAM models. First, in the MPC model with sublinear space per machine, our algorithms can compute an $O(f)$ approximation to SetCover in $\hat{O}(\sqrt{\log \Delta} + \log f)$ rounds[1] and a $O(\log \Delta)$ approximation in $O(\log^{3/2} n)$ rounds. Moreover, in the PRAM model, we give a $O(f)$ approximate algorithm using linear work and $O(\log n)$ depth. All these bounds improve the existing round complexity/depth bounds by a $\log^{\Omega(1)} n$ factor.

Moreover, our approach leads to many other new algorithms, including improved algorithms for the HypergraphMatching problem in the MPC model, as well as simpler SetCover algorithms that match the existing bounds.

2012 ACM Subject Classification Theory of computation → Massively parallel algorithms; Theory of computation → Shared memory algorithms; Theory of computation → MapReduce algorithms

Keywords and phrases approximate maximum matching, set cover, hypergraph matching, PRAM, massively parallel computation

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.19

Related Version *Full Version*: https://arxiv.org/abs/2408.13362

Funding *Michael Dinitz*: Supported in part by NSF awards CCF-1909111 and CCF-2228995.
Slobodan Mitrović: Supported by the Google Research Scholar and NSF Faculty Early Career Development Program No. 2340048.

1 Introduction

There is perhaps no more central and important problem in the area of approximation algorithms than SetCover. It has been a testbed for various algorithmic techniques that have become central in the field: greedy algorithms, deterministic and randomized rounding, primal-dual, dual fitting, etc. Due to its importance, ubiquity, and the fact that many different algorithmic techniques can be used, it is widely considered a "textbook problem" and, for example, has been used to illustrate the very basics of approximation

[1] We use the $\hat{O}(x)$ notation to suppress poly log x and poly log log n terms.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 19; pp. 19:1–19:23
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

algorithms [38, Chapter 1]. There are essentially two standard approximation bounds, both of which can be achieved through a number of different algorithms: an f-approximation, where f is the *frequency* (the maximum number of sets containing any given element), and an $H_\Delta = O(\log \Delta)$-approximation, where Δ is the maximum set size and H_k is the k'th harmonic number.[2]

Unsurprisingly, SetCover has also received significant attention in parallel and distributed models of computation. However, the simple sequential algorithms for SetCover are not "obviously" parallelizable, so new algorithms have been developed for these models. These lines of work range from classical complexity-theoretic models (e.g., showing that it can be approximated well in NC [7]), classical parallel models such as PRAMs [7, 34, 9], classical distributed models such as LOCAL [29, 28], and modern models such as MapReduce and Massively Parallel Computation (MPC) [36, 3]. Much of this work has been model-focused rather than model-independent, and ideas and techniques from one model can only sometimes be transferred to a different model.

In this paper, we introduce a new, simple, and model-independent technique for solving unweighted SetCover in parallel settings. Our technique, which involves careful independent random sampling of either the sets or elements, yields both a $(1+\epsilon)f$-approximation and a $(1+\epsilon)H_\Delta$-approximation and can be efficiently instantiated in multiple models of computation, including the MPC and PRAM models. Moreover, it can also be extended to solve the approximate HypergraphMatching problem in unweighted graphs. By applying our technique, we obtain efficient algorithms for SetCover and HypergraphMatching in MPC and PRAM models, which either improve upon or (essentially) match state-of-the-art algorithms for the problems. Importantly, our technique provides a unified and model-independent approach across HypergraphMatching and two variants of SetCover, and can be efficiently implemented in two fundamental models of parallel computation.

Our algorithms are obtained by parallelizing two classic f- and $O(\log \Delta)$-approximate SetCover algorithms. The f-approximate algorithm repeatedly picks an uncovered element and adds all sets containing it to the solution. The $O(\log \Delta)$-approximate in each step simply adds to the solution the set that covers the largest number of uncovered points.

Even though our parallelization of these algorithms is surprisingly direct, to the best of our knowledge, it has not been analyzed prior to our work. At a high level, our algorithms perform independent random sampling to find a collection of sets to be added to the solution, remove all covered elements from the instance, and then repeat. By combining the random sampling-based approach with modern techniques in parallel algorithms, we are able to give state-of-the-art bounds.

1.1 Our Contribution

We now present the main contributions of the paper. We study the unweighted version of SetCover. To formulate the bounds we obtain, we assume the SetCover problem is represented by a bipartite graph, in which vertices on one side represent the sets, and vertices on the other side represent elements to be covered. Edges connect elements with all sets that they belong to. We use Δ to denote the maximum degree of a vertex representing a set, and f to denote the maximum degree of a vertex representing an element. We use n to denote the number of vertices in the graph (equal to the number of sets plus the number of elements) and m to denote the number of edges (the total size of all sets).

[2] This result is often given as an $O(\log n)$-approximation since $\Delta \leq n$, but H_Δ is a technically stronger bound.

Table 1 Round complexity of SetCover algorithms in the Massively Parallel Computation model. We use n to denote the number of vertices in the graph (which is equal to the number of sets plus the number of elements) and m to denote the number of edges (the total size of all sets). $\delta \in (0,1)$ is a constant. In [22] $\phi \in (0,1]$ is any value satisfying $m \leq n^{1+\phi}$ and $c < \phi$ controls the amount of space per machine. We use ‡ to denote that a bound holds with high probability.

Ref.	Space/Machine	Total Space	Approx. Factor	Det.	Round Complexity
[9]	$O(n^\delta)$	$O(m)$	$(1+\epsilon)H_\Delta$	No	$O(\log^2 n)^\ddagger$
Here	$O(n^\delta)$	$\tilde{O}(m)$	$(1+\epsilon)H_\Delta{}^\ddagger$	No	$\hat{O}(\log\Delta \cdot \sqrt{\log f})^\ddagger$
Here	$O(n^\delta)$	$\tilde{O}(m)$	$(1+\epsilon)f^\ddagger$	No	$\hat{O}(\sqrt{\log\Delta})^\ddagger$
[6]	$O(n^\delta)$	$O(m)$	$f+\epsilon$	Yes	$O(\log\Delta/\log\log\Delta)$
[16]	$O(n^\delta)$	$O(m)$	$(1+\epsilon)f$	Yes	$O(\log(f\Delta)/\log\log(f\Delta))$
[3]	$\tilde{O}(n)$	$\tilde{O}(m)$	$O(\log n)^\ddagger$	No	$O(\log n)^\ddagger$
Here	$\tilde{O}(n)$	$\tilde{O}(m)$	$(1+\epsilon)H_\Delta{}^\ddagger$	No	$O(\log\Delta)^\ddagger$
[22]	$O(fn^{1+c})$	$O(m)$	f	No	$O((\phi/c)^2)$

We start by presenting our results in the Massively Parallel Computation (MPC) model [25, 20, 4, 1]. MPC computation proceeds in *synchronous* rounds over M machines. We assume that the input to the computation is partitioned arbitrarily across all machines in the first round. Each machine has a local space of η bits. In one round of computation, a machine first performs computation on its local data. Then, the machines can communicate by sending messages: each machine can send messages to any other machine. The messages sent in one round are delivered at the beginning of the next round. Hence, within a round, the machines, given the messages received in this round, work entirely independently. Importantly, the total size of the messages sent or received by a machine in a given round is at most η bits.

In the context of graph algorithms, there are three main regimes of MPC defined with respect to the relation of the available space on each machine η to the number of vertices of the graph n. In the *super-linear* regime, $\eta = n^{1+c}$ for a constant $0 < c < 1$. The *nearly-linear* regime requires $\eta = n \operatorname{poly}\log n$. Finally, the most restrictive and challenging *sub-linear* regime requires $\eta = n^c$. In all the regimes, we require that the total space of all machines is only a $\operatorname{poly}\log n$ factor larger than what is required to store the input.

In our definition of the SetCover problem, the number of vertices is the number of sets plus the number of elements. We note that some SetCover algorithms in the linear space regime (both prior and ours) only require space near-linear in the number of sets plus sublinear in the number of elements, but we use a single parameter for simplicity.

SetCover in MPC

Our first result is a set of improved MPC algorithms for SetCover.

▶ **Result 1.** *Let $\epsilon \in (0, 1/2)$ be a constant. Denote by f the maximum number of sets an element appears in, and by Δ the largest set size. Then, SetCover can be solved in MPC with the following guarantees:*

- *$(1+\epsilon)H_\Delta$-approximation in $\hat{O}\left(\operatorname{poly}(1/\epsilon) \cdot \log\Delta \cdot \sqrt{\log f}\right)$ rounds in the sub-linear regime,*
- *$(1+\epsilon)H_\Delta$-approximation in $O(\log\Delta)$ rounds in the nearly-linear regime,*
- *$(1+\epsilon)f$-approximation in $\hat{O}\left(\operatorname{poly}(1/\epsilon) \cdot \left(\sqrt{\log\Delta} + \log f\right)\right)$ rounds in the sub-linear regime.*

The algorithms use $\tilde{O}(m)$ total space, and the round complexities hold with high probability.

Table 2 Parallel cost bounds (work and depth) of f-approximate SetCover algorithms in the CRCW PRAM. m denotes the sum of the sizes of all sets (or the number of edges in the bipartite representation of SetCover), n denotes the number of elements, f denotes the maximum number of sets any element is contained in, and $\epsilon \in (0, 1/2)$ is an arbitrary constant. We use * to denote that a bound holds in expectation, and ‡ to denote that a bound holds with high probability.

Ref.	Approx. Factor	Det.	Work	Depth	Notes
[27]	$(1+\epsilon)f$	Yes	$O(fm)$	$O(f \log^2 n)$	
[28]	2	No	$O(m)^*$	$O(\log n)^\ddagger$	For weighted instances with $f = 2$.
Here	$(1+\epsilon)f^*$	No	$O(m)$	$O(\log n)$	

Before our work, the best-known round complexity for the $(1+\epsilon)H_\Delta$ SetCover in the sub-linear regime was $O(\log \Delta \cdot \log f)$; this complexity is implicit in [7]. Our algorithm improves this bound by a $\sqrt{\log f}$ factor. In the nearly-linear space regime, it is possible to achieve $O(\log n)$-approximation in $O(\log n)$ rounds by building on [3]. It is unclear how to transfer this approach to the sub-linear regime. We improve the approximation ratio to H_Δ, which is better, especially when $\Delta \ll n$.

In terms of $(1+\epsilon)f$-approximation, the most efficient SetCover algorithm in MPC follows by essentially a direct adaption of the Congest/Local $O(\log \Delta / \log\log \Delta)$ round algorithms in [6, 16] to MPC. Hence, for $f \leq 2^{O(\sqrt{\log n})}$, our work improves the MPC round complexity nearly quadratically.

SetCover in PRAM

Since our main algorithmic ideas are model-independent, they also readily translate to the PRAM setting, giving a new result for $(1+\epsilon)f$-approximate SetCover that improves over the state-of-the-art, and a streamlined $(1+\epsilon)H_\Delta$-approximation algorithm for SetCover [9].

▶ **Result 2.** *Let $\epsilon \in (0, 1/2)$ be an absolute constant. Let f be the maximum number of sets an element appears in, and let Δ be the largest set size. Then, SetCover can be solved in CRCW PRAM with the following guarantees:*

- *$(1+\epsilon)f$-approx. in expectation with* deterministic *$O(n+m)$ work and $O(\log n)$ depth.*
- *$(1+\epsilon)H_\Delta$-approx. in expectation with* deterministic *$O(n+m)$ work and $O(\log^2 n \log\log n)$ depth.*

In the context of $(1+\epsilon)f$-approximation, our result improves the state-of-the-art [27] total work by f while depth is improved by an $f \log n$ factor. For $(1+\epsilon)H_\Delta$-approximation, our result obtaining deterministic $O(\log^2 n \log\log n)$ depth and providing the approximation guarantee in expectation should be compared to the state-of-the-art PRAM algorithm of Blelloch, Peng, and Tangwongsan [9], which provides a depth guarantee in expectation and a worst-case guarantee for the approximation ratio. While the expected depth bound reported in [9] is $O(\log^3 n)$, we believe it can be improved to $O(\log^2 \log\log n)$ using some of the implementation ideas in our PRAM algorithm (see the full version for more discussion). A more detailed comparison between the prior and our results in PRAM is given in Table 2.[3]

[3] Our algorithms provide approximation in expectation. Nevertheless, this can be lifted to "with high probability" guarantees by executing $O(\log n/\epsilon)$ independent instances of our algorithm and using the smallest set cover. It incurs an extra $O(\log n/\epsilon)$ factor in the total work while not affecting the depth asymptotically.

HypergraphMatching in MPC

Finally, we also obtain an improved MPC algorithm for finding hypergraph matchings, i.e., for finding matchings in graphs where an edge is incident to (at most) h vertices.

▶ **Result 3.** *Let $\epsilon \in (0, 1/2)$ be an absolute constant. There is an MPC algorithm that, in expectation, computes a $(1-\epsilon)/h$ approximate maximum matching in a rank h hypergraph in the sub-linear space regime. This algorithm succeeds with high probability, runs in $\hat{O}\left(\text{poly}(1/\epsilon) \cdot \left(h^4 + h \cdot \sqrt{\log \Delta}\right)\right)$ MPC rounds and uses a total space of $\tilde{O}\left(m\right)$.*

Prior work [21] shows how to solve HypergraphMatching in rank h hypergraphs in $O(\log n)$ rounds in the *nearly-linear* space regime. So, for $h \in O(1)$, Result 3 improves quadratically over the known upper bound and, in addition, extends to the sub-linear space regime at the cost of slightly worsening the approximation ratio. For simple graphs, i.e., when $h = 2$, the work [19] already provides $\tilde{O}(\sqrt{\log \Delta})$ round complexity algorithm for computing $\Theta(1)$-approximate, and also maximal, matching. Nevertheless, our approach is arguably simpler than the one in [19] and, as such, lands gracefully into the MPC world.

1.2 Further Related Work

SetCover in the MPC Model

Both SetCover and VertexCover, i.e., SetCover with $f = 2$, have been extensively studied in the MPC model. Stergiou and Tsioutsiouliklis [36] studied the SetCover problem in MapReduce and provided an empirical evaluation. Their main algorithm is based on bucketing sets to within a $(1+\epsilon)$ factor with respect to the set sizes and then processing all the sets within the same bucket on one machine. Their algorithm, when translated to the MPC model, runs in $O(\log \Delta)$ iterations, but does not come with a bound on the required space per machine, which in the worst case can be linear in the input size.

Harvey, Liaw, and Liu [22] studied weighted VertexCover and SetCover in the MPC model and obtained results for both f and $(1+\epsilon)H_\Delta$-approximation. Their results exhibit a tradeoff between the round complexity and the space per machine. For f-approximation, they gave a $O((\phi/c)^2)$ round algorithm with space per machine $O(fn^{1+c})$ by applying filtering [30] to a primal dual algorithm. When the space per machine is nearly-linear, i.e., $c = O(1/\log n)$, this approach results in $O(\phi^2 \log^2 n)$ rounds, which is quadratically slower than our algorithm.

Bateni, Esfandiari, and Mirrokni [3] developed a MapReduce algorithm for the k-cover problem that uses $\tilde{O}(n)$ space per machine. In this problem, one is given an integer k and is asked to choose a family of at most k sets that cover as many elements as possible. The problem, since it is a submodular maximization under cardinality constraint, admits a $\Theta(1)$-approximation. Their algorithm can be turned into an $O(\log n)$-approximate one for SetCover that uses $O(\log n)$ MPC rounds as follows. Assume that k is the minimum number of sets that covers all the elements; this assumption can be removed by making $O(\log n/\epsilon)$ guesses of the form $k = (1+\epsilon)^i$. Then, each time [3] is invoked, it covers a constant fraction of the elements. So, repeating that process $O(\log n)$ times covers all the elements using $O(k \cdot \log n)$ many sets. Our result provides tighter approximation and, when $\Delta \ll n$, also lower round complexity.

Since k-cover is a submodular maximization problem, the work [32] yields $O(\log n)$ MPC round complexity and $O(\log n)$ approximation for SetCover. In the context of k-cover or SetCover, it is worth noting that the algorithm of [32] sends $\Theta(\sqrt{nk})$ sets to a machine. It is unclear whether all those sets can be compressed to fit in $O(n)$ or smaller memory.

Ghaffari and Uitto [19] developed a $\tilde{O}(\sqrt{\log \Delta})$ round complexity algorithm for VertexCover in the sub-linear space regime. They first compute a maximal independent set, which is then used to obtain a maximal matching in the corresponding line graph. Finally, by outputting the endpoints of the edges in that maximal matchings, the authors provide a 2-approximate VertexCover. Our algorithm has a matching round complexity while, at the same time, it is arguably simpler. For both f-approximation and H_Δ-approximation, we are unaware of any MPC algorithms that run in the sub-linear space regime. However, we note that the PRAM algorithm of Blelloch, Peng, and Tangwongsan [9] can be simulated in this setting to obtain a round complexity of $O(\log^2 n)$ with $O(m)$ total space.

f-Approximate SetCover in PRAM

The first f-approximation algorithms for SetCover in the sequential setting are due to Hochbaum [23]. In the unweighted case, we can sequentially obtain an f-approximation in $O(m)$ work by picking any element, adding all of $\leq f$ sets containing it to the cover, and removing all newly covered elements. For parallel algorithms aiming for f-approximation, Khuller, Vishkin, and Young [27] gave the first parallel $(1+\epsilon)f$-approximation for weighted SetCover that runs in $O(fm \log(1/\epsilon))$ work and $O(f \log^2 n \log(1/\epsilon))$ depth. Their method uses a deterministic primal-dual approach that in each iteration raises the dual values $p(e)$ on every uncovered element e until the primal solution, which is obtained by rounding every set s where $\sum_{e \in s} p(e) \geq (1-\epsilon)w(s)$, is a valid set cover. Their work analysis bounds the total number of times an element is processed across all $O(f \log n)$ iterations by m, giving a total work of $O(f \cdot m)$, which is not work-efficient. Their algorithm also has depth linear in f, which means that the number of iterations of their algorithm can be as large as $O(\log^2 n)$ for $f \leq \log n$, and the depth therefore as large as $O(\log^3 n)$.

For weighted VertexCover, Koufogiannakis and Young [28] gave an elegant 2-approximation that runs in $O(m)$ work in expectation and $O(\log n)$ depth. They generalize their algorithm to work for f-approximate weighted SetCover in the *distributed* setting using Linial-Saks decomposition [33]; however, this does not imply an NC or RNC algorithm when $f > 2$.[4]

Unlike the deterministic $(1+\epsilon)f$-approximation of Khuller et al. [27], our algorithm is randomized and produces a set cover with the same approximation guarantees in expectation. By contrast, our algorithm is easy to understand, analyze (with Lemma 3 as a given) and argue correctness. Our algorithm is well suited for implementation and has small constant factors, since every element set or element and their incident edges are processed *exactly once* when the element is sampled or when the set is chosen. We note that our algorithm also implies that $(1+\epsilon)f$-approximate SetCover is in RNC^1 for any f; the work of [27] only implies this result for $f = O(1)$.

Matching and HypergraphMatching in the Massively Parallel Computation Model

The study of approximate matchings in MPC was initiated by Lattanzi et al. [30], who developed an $O(1)$ round algorithm for finding a maximal matching when the space per machine is $n^{1+\mu}$, for any constant $\mu > 0$. In the linear space regime, a line of work [14, 17, 2, 5]

[4] NC contains all problems that admit log-space uniform circuits of polynomial size and poly-logarithmic depth and is the primary complexity class of interest when designing parallel algorithms. RNC extends NC by allowing the circuit access to randomness. By known simulation results [26], polynomial work and poly-logarithmic depth (randomized) PRAM algorithms also imply membership in NC (RNC).

culminated in $O(\log\log n)$ MPC round complexity. In the sublinear space regime, Ghaffari and Uitto [19] developed a method that finds a maximal matching in $\tilde{O}(\sqrt{\log n})$ rounds. When each machine has at least $O(nr)$ space, Hanguir and Stein [21] show how to find a maximal matching in r-hypergraph in $O(\log n)$ MPC rounds. Their approach follows the filtering idea developed in [30]. Our work does not only provide nearly quadratically lower round complexity compared to [21], but it also extends to the sub-linear space regime.

H_Δ-approximate SetCover in PRAM

Sequentially, H_Δ-approximate SetCover can be solved in $O(n+m)$ work by repeatedly selecting the set incident to the largest number of uncovered elements. The first parallel approximation algorithm for SetCover was due to Berger, Rompel and Shor [7], who gave a $(1+\epsilon)H_\Delta$-approximation that runs in $O(m\log^5 n)$ work and $O(\log^5 n)$ depth whp. Their algorithm buckets the sets based on their sizes into $O(\log\Delta)$ buckets. It then runs $O(\log f)$ subphases, where the j-th subphase ensures that all elements have degrees at most $(1+\epsilon)^j$ (the subphases are run in decreasing order). Each subphase performs $O(\log n)$ steps that work by either selecting sets that cover a constant fraction of certain large edges or otherwise independently sampling the remaining sets with probability $(1+\epsilon)^{-j}$. Our approach also uses independent sampling but does not require handling two cases separately. As a result, our approach can be implemented efficiently by fixing the random choices upfront (see Section 3.1). Subsequent work by Rajagopalan and Vazirani [34] improved the work and depth, obtaining a parallel primal-dual algorithm with $O(m\log^3 n)$ work and $O(\log^3 n)$ depth with high probability, but a weaker approximation guarantee of $2(1+\epsilon)H_\Delta$.

More recently, Blelloch, Peng and Tangwongsan [9] revisited parallel approximate SetCover with the goal of designing work-efficient algorithms. Their algorithm achieves a $(1+\epsilon)H_\Delta$-approximation in $O(m)$ expected work and $O(\log^3 n)$ depth with high probability on the CRCW PRAM. They propose a general primitive inspired by the approach of [34] called a Maximal Nearly-Independent Set (ManIS), which, given a collection of sets chooses a subset of them while ensuring that the chosen sets are (1) nearly independent and thus do not have significant overlap, and (2) maximal, so that any unchosen sets have significant overlap with chosen ones. Blelloch, Simhadri, and Tangwongsan [10] later studied the algorithm in the Parallel Cache Oblivious model, and provided an efficient parallel implementation.

Compared to this prior work, we obtain a streamlined $(1+\epsilon)H_\Delta$-approximate algorithm that shares some ideas with the previously discussed algorithms. We also bucket the sets by size, and like [34, 9] each round finds a subset of sets with low overlap; the main difference is that our method is arguably simpler. Our algorithm is also potentially very efficient in practice, since after we fix the randomness up-front (see Section 3.1), we process every set in a bucket exactly once, unlike other implementations of ManIS which can process a set within a bucket potentially many times [15]. Overall, our algorithm is work-efficient and runs in $O(\log^2 n\log\log n)$ depth on the CRCW PRAM. Although this is an improvement over known depth bounds for PRAM algorithms, one can obtain similar bounds (in expectation) for the algorithm of [9] by applying similar PRAM techniques.We also note that both algorithms achieve $O(\log^3 n)$ depth in the binary-forking model [8], and no parallel H_Δ-approximate algorithms exist with $o(\log^3 n)$ depth in this model. Experimentally comparing our algorithm with existing implementations of [9] is an interesting direction for future work.

1.3 Outline

The rest of the paper is organized as follows. In Section 2 we introduce notation that we use in the paper. Section 3 contains a technical overview of our results. In particular, it describes our algorithms and outlines how they can be analyzed and efficiently implemented in the MPC and PRAM models. Then, in Section 4 we provide the approximation analysis of our basic algorithms. Finally, in Appendix A we provide the formal analysis of the random process which we use to model our algorithms. Due to space constraints, the remaining details, including the detailed descriptions of the MPC and PRAM algorithms are deferred to the full version of this paper.

2 Preliminaries

In the SetCover problem, we have a collection of *elements* T and a family of *sets* S, which we can use to cover elements of T. We represent an instance of the problem with a bipartite graph $G = ((S \cup T), E)$, where $st \in E$ if and only if element t belongs to the set s. For a vertex $x \in S \cup T$ we use $N(x)$ to denote the set of its neighbors. Since G is bipartite, $x \in S$ implies $N(x) \subseteq T$ and $x \in T$ implies $N(x) \subseteq S$. In particular, for $x \in S$, $|N(x)|$ is the size of the set x.[5] We use Δ to denote the maximum set size (i.e., the maximum degree of any vertex in S) and f to denote the largest number of sets that contain any element (the maximum degree of any vertex in T). Note that some of our algorithms modify the input graphs along the way, but we assume Δ and f to be constant and refer to the corresponding quantities in the input graph.

The VertexCover problem is defined as follows. The input is an undirected graph $G = (V, E)$ and the goal is to find the smallest set $C \subseteq E$ such that each edge has at least one endpoint in C. We note that this problem is equivalent to the SetCover problem in which each element belongs to exactly 2 sets, except that the graph representing an instance is constructed a bit differently.

In the HypergraphMatching problem, the input is a hypergraph G consisting of a set of vertices V and a set of edges E. Each edge is a nonempty subset of V. The *rank* of a hypergraph G is the maximum size of any edge. In the HypergraphMatching problem the goal is to find a subset $M \subseteq E$ which contains pairwise disjoint edges and has maximum possible size. As opposed to SetCover, this is a maximization problem, and thus we say that the solution M to the HypergraphMatching problem is α-approximate, for $\alpha \in (0, 1]$, when $|M| \geq \alpha \cdot |\mathrm{OPT}|$, where OPT is an optimal solution to the HypergraphMatching problem. The Matching problem is the HypergraphMatching problem in simple graphs, i.e., in graphs with all edges of size 2.

Notation. We use $\tilde{O}(x)$ to hide logarithmic factor in x, i.e., $\tilde{O}(x)$ denotes $O(x \cdot \operatorname{poly}\log x)$. Throughout this paper, we use n to refer to the number of vertices and m to refer to the number of edges of an input graph. When it is stated that a guarantee holds "with high probability", or whp for short, it means that it holds with probability $1 - 1/n^c$, where c is a constant. In our proofs with whp guarantees, c can be made arbitrarily large by paying a constant factor in the round, space, total work, or depth complexity. Hence, we often omit the exact value of c.

[5] Technically, $|x|$ is also the size of the set x. However, in our algorithms, we repeatedly remove some elements from T (together with their incident edges), and so we use $|N(x)|$ to make it clear that we refer to the *current* size of the set.

Probability tools. In our analysis, we extensively apply the following well-known tool from probability.

▶ **Theorem 1** (Chernoff bound). *Let $X_1, \ldots, X_k$ be independent random variables taking values in* $[0,1]$. *Let* $X \overset{\text{def}}{=} \sum_{i=1}^{k} X_i$ *and* $\mu \overset{\text{def}}{=} \mathbb{E}[X]$. *Then,*

(A) *For any* $\delta \in [0,1]$ *it holds* $\Pr\left[X \le (1-\delta)\mu\right] \le \exp\left(-\delta^2\mu/2\right)$.

(B) *For any* $\delta \in [0,1]$ *it holds* $\Pr\left[X \ge (1+\delta)\mu\right] \le \exp\left(-\delta^2\mu/3\right)$.

Work-Depth Model. We study our algorithms in the shared-memory setting using the concurrent-read concurrent-write (CRCW) parallel random access machine model (PRAM). We state our results in terms of their work and depth. The *work* of a PRAM algorithm is equal to the total number of operations required, and the *depth* is equal to the number of time steps required [24]. Algorithms with work W and depth D can be scheduled to run in $W/P + O(D)$ time [24, 11] on a P processor machine. The main goal in parallel algorithm design is to obtain work-efficient algorithms with low (ideally poly-logarithmic) depth. A *work-efficient* algorithm asymptotically requires the same work as the fastest sequential algorithm. Since the number of processors, P, is still relatively small on modern multicore machines, minimizing W by designing work-efficient algorithms is critical in practice. Our algorithms make use of several PRAM primitives, including parallel prefix sum [24], parallel integer sort [35], and approximate prefix sums [23].

3 Technical Overview

In this section we demonstrate the main ideas behind our results. We start by presenting our sequential algorithms for the SetCover problem. For any set X and probability $p \in [0,1]$ we write Sample(X, p) to denote a procedure that returns a random subsample of X in which each element of X is included independently with probability p. Our algorithms work by repeatedly sampling sets or elements independently using a sequence of probabilities p_i, which is defined as follows for any $\epsilon > 0$.

$$b \overset{\text{def}}{=} \lceil \log(2+2\epsilon)/\epsilon \rceil \tag{1}$$

$$p_i \overset{\text{def}}{=} (1+\epsilon)^{-\lceil i/b \rceil} \qquad \text{for any } i \in \mathbb{N}. \tag{2}$$

Throughout the paper, we use log to denote the natural logarithm function. We can now present our algorithms for SetCover, which are given as Algorithm 1 and Algorithm 2.

Algorithm 1 is a natural parallelization of the sequential f-approximate algorithm. Instead of picking one element at a time, we sample multiple elements at random and add to the solution the sets containing them. The sampling probability is slowly increased in each step (or, more precisely, every $O(1/\epsilon)$ steps). Algorithm 2 in turn parallelizes the $O(\log \Delta)$ approximate algorithm. The outer loop iterates over different set sizes (rounded to the power of $1+\epsilon$) starting from the largest ones. For a fixed set size, the inner loop adds to the solution a uniformly random sample of sets, again slowly increasing the sampling probability.

We start by analyzing Algorithm 1. Clearly, the algorithm runs in $O(\log n)$ iterations. Iteration i samples each element independently with probability p_i and adds all sets covering the sampled elements to the solution. Then, all chosen sets and elements that became covered are removed. Crucially, the sampling probability in the first step is at most $(1+\epsilon)^{-\lceil \log_{1+\epsilon}(\Delta/\epsilon)\rceil} \le \epsilon/\Delta$, which implies that the expected number of elements sampled within each set is at most ϵ. Moreover, the sampling probability increases very slowly, as it increases by a $1+\epsilon$ factor every b steps.

Algorithm 1 $(1+\epsilon)f$-approximate algorithm for SetCover.

```
function SetCover($G, \epsilon$)                                  ▷ $G = (S \cup T, E)$
    $C \leftarrow \emptyset$
    for $i = b\lceil \log_{1+\epsilon}(\Delta/\epsilon) \rceil$ down to 0 do
        $D \leftarrow$ Sample($T, p_i$)
        $C \leftarrow C \cup N(D)$
        Remove from $G$ all sets in $N(D)$ and all elements they cover
    return $C$
```

Algorithm 2 $(1+\epsilon)H_\Delta$-approximate algorithm for SetCover.

```
function SetCover($G, \epsilon$)                                  ▷ $G = (S \cup T, E)$
    $C \leftarrow \emptyset$
    for $j = \lfloor \log_{1+\epsilon} \Delta \rfloor$ down to 0 do
        for $i = b\lceil \log_{1+\epsilon}(f/\epsilon) \rceil$ down to 0 do
            $D \leftarrow$ Sample($\{s \in S \mid |N(s)| \geq (1+\epsilon)^j\}, p_i$)
            $C \leftarrow C \cup D$
            Remove from $G$ all sets in $D$ and all elements they cover
    return C
```

Let us now present the main ideas behind the analysis of the approximation ratio of the algorithm. For simplicity of presentation, let us consider the case when each element is contained in exactly two sets (which implies $f = 2$). In other words, we consider the VertexCover problem. Specifically, since the degree of each vertex of T is 2, we can dissolve vertices of T (equivalently, contract each such vertex into its arbitrary neighbor) and obtain a graph $H = (V, E)$ (where $V = S$) on which we would like to solve the VertexCover problem. We note that the solution and analysis of Algorithm 1 for VertexCover generalizes easily to the case of arbitrary f.

If we translate Algorithm 1 to an algorithm running on H, we see that it repeatedly samples a set of *edges* of H, and for each sampled edge e adds both endpoints of e to the solution, and removes both endpoints of e from H together with their incident edges. In order to prove the approximation guarantee, we show the following.

▶ **Lemma 2.** *Let D be the subset of T picked across all iterations of Algorithm 1. For each vertex v, $\mathbb{E}[\deg_D(v))] \leq 1 + O(\epsilon)$.*

Here $\deg_D(v)$ denotes the number of elements of D contained in v. We prove this lemma formally in Section 4 (see Lemma 6).

Notice that when an element $x \in T$ is sampled to D in Algorithm 1, *all* the sets containing x are added to C. So if w sampled elements belong to the same set, then the algorithm could add $\Theta(wf)$ many sets, although only one of the sets suffices to cover all the w sampled elements. Intuitively, Lemma 2 states that the value of w is at most $1 + O(\epsilon)$ in expectation, which we turn into an approximation guarantee in Lemma 7.

To prove Lemma 2, we model the sampling process in the algorithm as follows. Fix a vertex $v \in V$. Let A be the set of edges incident to v in G. Algorithm 1 runs a sequence of $b\lceil \log_{1+\epsilon}(\Delta/\epsilon) \rceil + 1$ steps, indexed by $b\lceil \log_{1+\epsilon}(\Delta/\epsilon) \rceil, \ldots, 0$. Note that the step indices are *decreasing*. Moreover, $\Delta \geq |A|$, since Δ is the maximum vertex degree in G. In step i, each element of A is sampled independently with probability p_i. As soon as at least one element of A is sampled, v is added to the cover. When this happens all elements of A are deleted and the random process stops. Moreover, even if no element of A is sampled, due to other random choices of the algorithm some elements of A may get deleted. In particular, when a neighbor w of v is added to the set cover, the edge wv is deleted from A. Hence, it is possible that all elements of A are deleted before any of them is sampled.

To analyze this we introduce the following *set sampling process*, henceforth denoted as SSP, which gives a more abstract version of the above sampling. The analysis of this process forms our core sampling lemma, which will be useful not just for Lemma 2 but to analyze all of our algorithms. Let A be a fixed set and k be an integer. The process proceeds in $k+1$ steps indexed $k, k-1, \ldots, 0$ and constructs a family of sets $A = A_k \supseteq A_{k-1} \supseteq \cdots \supseteq A_1 \supseteq A_0$ as well as a family $R_k, \ldots, R_0$, such that $R_i \subseteq A_i$. In each step i ($k \geq i \geq 0$) we first construct a set A_i. We have that $A_k = A$, and for $i \in [0, k)$ the set $A_i \subseteq A_{i+1}$ is constructed by a (possibly randomized) *adversary*, who is aware of the sets A_j and R_j for $j > i$. In our analysis of SSP, the goal is to argue that certain guarantees hold regardless of what the adversary does. After the adversary constructs A_i, we sample $R_i = \textsc{Sample}(A_i, p_i)$.

We note that we assume that the updates are adversarial to simplify the overall proof. This makes our claims about SSP more robust, and analyzing SSP with an adversary does not introduce significant complications.

For $i \in [0, k]$, we define $n_i \overset{\text{def}}{=} |A_i|$. Whenever we apply SSP we have that $k \geq b\lceil \log_{1+\epsilon}(n_k/\epsilon) \rceil$, and for simplicity we make this assumption part of the construction. Note that this condition simply ensures that the initial sampling probability is at most ϵ/n_k. Finally, we let z be the maximum index such that $R_z \neq \emptyset$. We stress that the SSP steps are indexed in *decreasing order*, and hence z is the index of the *first* step such that R_z is nonempty. If all R_i are empty, we set $z = -1$ and assume $R_{-1} = \emptyset$. We say that z is the step when the SSP stops.

Observe that in order to analyze the properties of the set of sampled edges in Algorithm 1, it suffices to analyze the properties of the set R_z. Our main lemma analyzing SSP is given below. It captures the single property of SSP which suffices to prove the approximation ratio of both Algorithm 1 and Algorithm 2. In particular, it directly implies Lemma 2.

▶ **Lemma 3.** *Consider the SSP using any adversary and $\epsilon > 0$. Then, $\mathbb{E}[|R_z|] \leq 1 + 4\epsilon$.*

Let us now describe the intuition behind the proof of Lemma 3. To simplify presentation, let us assume that the sets $A_0, \ldots, A_k$ are fixed upfront (i.e., before any set R_i is sampled). We show in Observation 11 that if we are interested in analyzing the properties of R_z, this can be assumed without loss of generality. Observe that as long as the process executes steps where $p_i \cdot n_i \leq \epsilon$, the desired property holds. Indeed, with this assumption we have that $\mathbb{E}[|R_i| \mid R_i \neq \emptyset] \leq 1 + \epsilon$. This is because even if one element is sampled, the expected size of the sample among all remaining elements is at most ϵ (for a formal proof, see Claim 16).

In order to complete the proof, we show that reaching a step where $p_i \cdot n_i \gg \epsilon$ is unlikely. Specifically, the value of $p_i \cdot n_i$ can increase very slowly in consecutive steps, as p_i increases only by $(1+\epsilon)$ factor every b steps, and n_i can only decrease. By picking a large enough value of b, we can ensure that the process most likely stops before $p_j \cdot n_j$ becomes large, i.e., the expected value of $p_z \cdot n_z$ is $O(\epsilon)$. Indeed, in each step where $p_i \cdot n_i \geq \epsilon$, the process stops with probability $\Omega(\epsilon)$. Hence, if we repeat such a step roughly $1/\epsilon$ times (which can be achieved by tweaking b), the process will stop with constant probability (independent of ϵ). In the end we fix b, such that the probability of $p_i \cdot n_i$ increasing by a factor of $1 + \epsilon$ is at most $1/(2 + 2\epsilon)$. As a result, thanks to a geometric sum argument, the expected value of $p_z \cdot n_z$ is $O(\epsilon)$, which implies Lemma 3.

Algorithm 3 $(1+\epsilon)f$-approximate algorithm for SETCOVER.

```
function SETCOVER($G, \epsilon$)                                   ▷ $G = (S \cup T, E)$
    $C \leftarrow \emptyset$
    $k \leftarrow b\lceil \log_{1+\epsilon}(\Delta/\epsilon) \rceil$
    $B_i \leftarrow \emptyset$ for all $i \in [0, k]$
    for each element $t \in T$ do
        Sample $X_t \in [0, k]$, where $P(X_t = i) = \tilde{p_i}$ and add $t$ to $B_{X_t}$
    for $i = k$ down to $0$ do
        $D \leftarrow$ all elements of $B_i$ which are not marked
        $C \leftarrow C \cup N(D)$
        Remove from $G$ all sets in $N(D)$ and mark all elements they cover
    return $C$
```

3.1 Fixing the Random Choices Upfront

In order to obtain efficient implementations of our algorithms, we reformulate them into equivalent versions where the sampling happens upfront. Specifically, consider the main loop of Algorithm 1. Observe that each element is sampled at most once across all iterations, since as soon as an element is sampled it is removed from further consideration. A similar property holds for each set across all iterations of the inner for loop of Algorithm 2. Moreover, in both cases, the probability of being sampled in a given iteration is fixed upfront and independent of the algorithm's actions in prior iterations. It follows easily that we can make these per-element or per-set random choices upfront. Specifically, let $k = b\lceil \log_{1+\epsilon}(\Delta/\epsilon) \rceil$. Then, Algorithm 1 executes $k+1$ iterations indexed $k, k-1, \ldots, 0$. We can randomly partition the input elements into $k+1$ *buckets* $B_k, \ldots, B_0$ using a properly chosen distribution and then in iteration i consider the elements of B_i which have not been previously removed as the sample to be used in this iteration.

Observe that since $p_0 = 1$ (see Equation (2)), each element that is not removed before the last step is sampled. Specifically, let $\tilde{p_0}, \ldots, \tilde{p_k}$ be a probability distribution such that $\tilde{p_i} = p_i \cdot \prod_{j=i+1}^{k}(1 - p_j)$. Observe that $\tilde{p_i}$ is the probability that an element should be put into bucket B_i.

Algorithm 3 shows a version of Algorithm 1 in which the random choices are made upfront. It should be clear that Algorithms 1 and 3 produce the same output. Moreover, an analogous transformation can be applied to the inner loop of Algorithm 2. The benefit of making the random choices upfront is twofold. In the MPC model, we use the sampling to simulate r iterations of the algorithms in $O(\log r)$ MPC rounds. The efficiency of this simulation crucially relies on the fact that we only need to consider the edges sampled within the phase and we can determine (a superset of) these edges upfront.

In the PRAM model, the upfront sampling allow us to obtain an improved work bound: instead of tossing a coin for each element separately in each iteration, we can bucket the elements initially and then consider each element in exactly one iteration. In order to bucket the elements efficiently we can use the following lemma.

▶ **Lemma 4** ([37]). *Let $r_0, r_1, \ldots, r_k$ be a sequence of nonnegative real numbers which sum up to 1. Let $X \rightarrow [0, k]$ be a discrete random variable, such that for each $i \in [k]$, $P(X = i) = r_i$. Then, there exists an algorithm which, after preprocessing in $O(k)$ time, can generate independent samples of X in $O(1)$ time.*

3.2 MPC Algorithms

Simulating Algorithm 3 in the sub-linear MPC model is a relatively straightforward application of the graph exponentiation technique [31, 19, 18, 13, 12]. For simplicity, let us again consider the VertexCover problem. We will show how to simulate $r = O(\sqrt{\log n})$ iterations of the for loop in only $O(\log r) = O(\log \log n)$ MPC rounds. Let us call these r iterations of the algorithm a *phase*. We first observe that to execute a phase we only need to know edges in the buckets corresponding to the iterations within the phase. Let us denote by G_r the graph consisting of all such edges. Moreover, let p be the sampling probability used in the first iteration of the phase. The crucial observation is that the maximum degree in G_r is $2^{\tilde{O}(\sqrt{\log n})}$ with high probability. This can be proven in three steps. First, we show that by the start of the phase the maximum degree in the original graph G drops to $O(1/p \cdot \log n)$ with high probability. Indeed, for any vertex v with a higher degree the algorithm samples an edge incident to v with high probability, which causes v to be removed. Second, we observe that the sampling probability increases to at most $p \cdot 2^{O(\sqrt{\log n})}$ within the phase, and so the expected number of edges incident to any vertex of G_r is at most $O(1/p \cdot \log n) \cdot p \cdot 2^{O(\sqrt{\log n})} = 2^{\tilde{O}(\sqrt{\log n})}$. Third, we apply a Chernoff bound.

At this point, it suffices to observe that running r iterations of the algorithm can be achieved by computing for each vertex v of G_r a subgraph S_v consisting of all vertices at distance $O(r)$ from v and then running the algorithm separately on each S_v. In other words, running r iterations of the algorithm is a $O(r)$ round Local algorithm. Computing S_r can be done using graph exponentiation in $\log r = O(\log \log n)$ MPC rounds using $2^{O(\sqrt{\log n}) \cdot r} = n^{\alpha}$ space per machine and $n^{1+\alpha}$ total space, where $\alpha > 0$ is an arbitrary constant.

The space requirement can also be reduced to $\tilde{O}(m)$. We now sketch the high-level ideas behind this improvement. We leverage the fact that if we sample each edge of an m-edge graph independently with probability p, then only $O(p \cdot m)$ vertices have an incident sampled edge, and we can ignore all the remaining vertices when running our algorithm. Hence, we only need to run the algorithm for $O(p \cdot m)$ vertices and thus have at least $S = m/O(p \cdot m) = \Omega(1/p)$ available space per vertex, even if we assume that the total space is $O(m)$. As argued above, with space per vertex S, we can simulate roughly $\sqrt{\log S}$ steps of the algorithm. In each of these steps, the sampling probability increases by a constant factor, so overall, it increases by a factor of $2^{\Omega(\sqrt{\log S})}$ across the $\sqrt{\log S}$ steps that we simulate. After repeating this simulation $t = \sqrt{\log S} = O(\sqrt{\log \Delta})$ times, the sampling probability increases by a factor of at least $2^{\Omega(t \cdot \sqrt{\log S})} = 2^{\Omega(\log S)} = S^{\Omega(1)}$. Overall, after roughly $O(\sqrt{\log \Delta})$ repetitions the space per vertex reduces from S to $S^{1-\Omega(1)}$. Similarly, the sampling probability increases from p to $p^{1+\Omega(1)}$. Hence, it suffices to repeat this overall process $\log \log \Delta$ times to simulate all $O(\log \Delta)$ steps.

3.3 PRAM algorithms

Algorithm 3 also almost immediately yields a work-efficient algorithm with $O(\log n)$ depth in the CRCW PRAM. Obtaining a work-efficient and low-depth implementation of Algorithm 2 is only a little more involved. One challenge is that the set sizes change as elements get covered. Since we run $O(\log n)$ steps per round, we can afford to exactly compute the sizes at the start of a round, but cannot afford to do so on every step without incurring an additional $O(\log n)$ factor in the depth. We first use the randomness fixing idea described in Section 3.1 to identify the step in the algorithm when a set will be sampled. Then, in every step, for the sets sampled in this step, we approximate the set sizes up to a $(1 + \delta)$

Algorithm 4 Algorithm for HypergraphMatching.

```
function HypergraphMatching($(G, \epsilon)$)                ▷ $G = (V, E)$
    $\Delta \leftarrow$ the maximum degree in $G$
    $C \leftarrow \emptyset$
    for $i = b\lceil \log_{1+\epsilon}(\Delta/\epsilon) \rceil$ down to $0$ do
        $D \leftarrow$ Sample($E(G), p_i$)
        $C \leftarrow C \cup D$
        Remove from $G$ all endpoints of edges in $D$
    return edges independent in $C$
```

factor, which can be done deterministically and work-efficiently in $O(\log\log n)$ depth and use these estimates in our implementation of Algorithm 2. The resulting algorithm still obtains a $(1+\epsilon)H_\Delta$-approximation in expectation while deterministically ensuring work efficiency and $O(\log^2 n \log\log n)$ depth.

3.4 HypergraphMatching in MPC

We show that our techniques for solving SetCover can be further applied to solve approximate HypergraphMatching. For the purpose of this high-level overview we consider the special case of approximate Matching in simple graphs, i.e., hypergraphs in which each edge has exactly 2 endpoints. Generalizing our approach to arbitrary hypergraphs does not require any additional ideas. Our algorithm for HypergraphMatching is shown as Algorithm 4, and works similarly to Algorithm 1. Specifically, if we consider the simple graph setting and the VertexCover problem, Algorithm 1 samples a set of edges of the graph and then returns the set of endpoints of these edges as the solution. Algorithm 4 also samples a set of edges, but the difference is in how it computes the final solution. Namely, it returns all sampled edges which are independent, i.e., not adjacent to any other sampled edge. Clearly, the set of edges returned this way forms a valid matching. To argue about its cardinality, we show that the number of edges that are returned is a constant factor of all edges that have been sampled. To this end, we show a second fact about the SSP, which says that any sampled element is *not* sampled by itself with only small constant probability.

▶ **Lemma 5.** *Let $a \in A_k$ and let $\epsilon \leq 1/2$. Then $P(|R_z| > 1 \mid a \in R_z) \leq 6\epsilon$.*

The high-level idea behind the proof of Lemma 5 is similar to the proof of Lemma 2: in the steps where the expected number of sampled elements is $\leq \epsilon$, the property follows in a relatively straightforward way. Moreover, we are unlikely to reach any step where the expected number of sampled elements is considerably larger, and so to complete the proof we also apply a geometric sum-based argument. With the above Lemma, the analysis of Algorithm 4 becomes straightforward and shows that the approximation ratio of the algorithm is $\frac{1-h\cdot 6\epsilon}{h}$ (see Lemma 10), where h is the rank of the hypergraph.

Algorithm 4 can be seen as a simplification of the "warm-up" algorithm of [19], which alternates between sampling edges incident to high-degree vertices and peeling high-degree vertices. Our algorithm simply samples from *all edges* and does not peel vertices. This makes the proof of the approximation ratio trickier since there is less structure to leverage. However, the simplification results in a straightforward application of round compression and enables extending the algorithm to hypergraphs.

$$\begin{array}{lll} \text{minimize} & \sum_{s \in S} x_s & \\ \text{subject to} & \sum_{s \ni t} x_s \geq 1 & \text{for } t \in T \\ & x_s \geq 0 & \text{for } s \in S \end{array} \qquad \begin{array}{lll} \text{maximize} & \sum_{t \in T} y_t & \\ \text{subject to} & \sum_{t \in s} y_t \leq 1 & \text{for } s \in S \\ & y_t \geq 0 & \text{for } t \in T \end{array}$$

Figure 1 LP relaxation of the SetCover LP (left) and its dual (right).

4 Approximation Analysis of the Algorithms

In this section, we analyze the approximation ratio and analysis of our algorithms. We note that the correctness of all algorithms is essentially immediate. Specifically, in Algorithm 1 and Algorithm 2 we only remove an element when it is covered, and in the last iteration of the inner **for** loop in both algorithms (which in Algorithm 1 is the only loop) the sampling probability is 1, so we add all remaining sets (in Algorithm 2, limited to the large enough size) to the solution. Similarly, Algorithm 4 clearly outputs a valid matching, thanks to the final filtering step in the **return** statement.

▶ **Lemma 6.** *Let $\bar{D}$ be the union of all elements picked in all iterations of Algorithm 1. For each set $s \in S$ we have $\mathbb{E}[|s \cap \bar{D}|] \leq 1 + 4\epsilon$.*

Proof. This follows from Lemma 3 applied to the set s. ◀

▶ **Lemma 7.** *Algorithm 1, called with $e' = \epsilon/4$, computes an $(1+\epsilon)f$-approximate solution to SetCover.*

Proof. The key property that we utilize in the analysis is stated in Lemma 6. The proof is a relatively simple generalization of the dual fitting analysis of the standard f-approximate SetCover algorithm. The generalization needs to capture two aspects: the fact that the property stated in Lemma 6 holds only in expectation and allows for a slack of 4ϵ.

We use the relaxation of the SetCover IP and its dual given in Figure 1. Let $\bar{D}$ be the union of all elements picked in all iterations of Algorithm 1. We construct a dual solution that corresponds to $\bar{D}$ as follows. First, for each $t \in \bar{D}$, we set $\bar{y}_t = 1/(1+4\epsilon)$, and for $t \notin \bar{D}$ we set $\bar{y}_t = 0$. Recall that Algorithm 1 returns a solution of size $|C|$. For any run of the algorithm we have $|C| \leq f \sum_{t \in T} \bar{y}_t(1+4\epsilon)$.

We now define a set dual of variables by setting $y_t = \mathbb{E}[\bar{y}_t]$ for each $t \in T$. This set forms a feasible dual solution, since for every $s \in S$ we have

$$\sum_{t \in s} y_t = \sum_{t \in s} \mathbb{E}[\bar{y}_t] = \mathbb{E}[|s \cap \bar{D}|/(1+4\epsilon)] \leq 1,$$

where in the last inequality we used Lemma 6. Moreover, we have

$$\mathbb{E}[|C|] \leq f \cdot \sum_{t \in T} y_t(1+4\epsilon),$$

which implies that the solution's expected size is at most $f(1+4\epsilon)$ times larger than a feasible dual solution. Hence, the lemma follows from weak LP duality. ◀

Now let us consider Algorithm 2.

▶ **Lemma 8.** *Algorithm 2 adds sets to the solution in batches. When a batch of sets D is added to the solution we have that (a) the residual size of each set in D is at most $(1+\epsilon)$ smaller than the maximum residual size of any set at that moment, and (b) for each newly covered element t, the expected number of sets in a batch that cover it is at most $(1+4\epsilon)$.*

Proof. Observe that for $i=0$ and the current value of j, each of the remaining sets of size $(1+\epsilon)^j$ or more is included in D. By applying this observation inductively, we see that each iteration of the outer loops starts with the maximum set size being less than $(1+\epsilon)^{j+1}$ and results in all sets of size at least $(1+\epsilon)^j$ being either added to the solution or removed from the graph. This implies claim (a). Claim (b) follows directly from Lemma 3. ◀

To bound the approximation guarantee of Algorithm 2, we show the following, which, similarly to the proof of Lemma 7, uses a dual-fitting analysis.

▶ **Lemma 9.** *Any algorithm that computes a valid SetCover solution and satisfies the property of Lemma 8, computes an $(1+\epsilon)(1+4\epsilon)H_\Delta$-approximate (in expectation) solution to SetCover.*

▶ **Lemma 10.** *Algorithm 4 ran on a rank h hypergraph in expectation computes a $\frac{1-h\cdot 6\epsilon}{h}$ approximate matching.*

Proof. Let $G=(V,E)$ be input to Algorithm 4, and let C' be the set C after the execution of the for-loop. Observe that $V(C')$ is a vertex cover of G: all the edges not covered by the time we reach $i=0$ are included in D and, so, in C.

We want also to lower-bound the size of independent edges in C'. Fix an edge e and consider a vertex $v \in e$. Once e is included in C, all the endpoints of e are removed from G. Hence, if e is not independent in C', then it is the case because, in the same iteration, an edge e' adjacent to e is also included in D. To upper-bound the probability of e' and e being included in D, we use Lemma 5. How do we use Lemma 5 in the context of Algorithm 4? For a fixed vertex v, A_i is the set of edges incident to v at the i-th iteration of the for-loop of Algorithm 4. In particular, the set $A_k = A$ defined in Appendix A equals all the edges of the input graph G containing v.

By Lemma 5, the probability of v being incident to more than one sampled edge is at most 6ϵ. Thus, by union bound, e and an edge adjacent to e are included in D with probability at most $h\cdot 6\epsilon$. Therefore, with probability $1-h\cdot 6\epsilon$ at least, a fixed edge in C' is independent. This implies that in expectation $|C'|(1-h\cdot 6\epsilon)$ edges in C' are independent and, so, Algorithm 4 outputs a matching that in expectation has size at least $|C'|(1-h\cdot 6\epsilon)$. Since there is a vertex cover of size $|V(C')| \le h|C'|$ at most, it implies that Algorithm 4 in expectation produces a $\frac{1-h\cdot 6\epsilon}{h}$-approximate maximum matching. ◀

References

1 Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *ACM Symposium on Theory of Computing (STOC)*, pages 574–583, 2014. doi:10.1145/2591796.2591805.

2 Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab Mirrokni, and Cliff Stein. Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1616–1635. SIAM, 2019. doi:10.1137/1.9781611975482.98.

3 MohammadHossein Bateni, Hossein Esfandiari, and Vahab Mirrokni. Optimal distributed submodular optimization via sketching. In *ACM International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 1138–1147, 2018. doi:10.1145/3219819.3220081.

4 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *Journal of the ACM (JACM)*, 64(6):1–58, 2017. doi:10.1145/3125644.

5 Soheil Behnezhad, Mohammad Taghi Hajiaghayi, and David G Harris. Exponentially faster massively parallel maximal matching. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1637–1649. IEEE, 2019.

6 Ran Ben Basat, Guy Even, Ken-ichi Kawarabayashi, and Gregory Schwartzman. Optimal distributed covering algorithms. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 104–106, 2019. doi:10.1145/3293611.3331577.

7 Bonnie Berger, John Rompel, and Peter W Shor. Efficient NC algorithms for set cover with applications to learning and geometry. *Journal of Computer and System Sciences*, 49(3):454–477, 1994. doi:10.1016/S0022-0000(05)80068-6.

8 Guy E Blelloch, Jeremy T Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 89–102, 2020. doi:10.1145/3350755.3400227.

9 Guy E Blelloch, Richard Peng, and Kanat Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 23–32, 2011. doi:10.1145/1989493.1989497.

10 Guy E Blelloch, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Parallel and i/o efficient set covering algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 82–90, 2012. doi:10.1145/2312005.2312024.

11 Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999. doi:10.1145/324133.324234.

12 Sebastian Brandt, Manuela Fischer, and Jara Uitto. Breaking the linear-memory barrier in MPC: Fast MIS on trees with strongly sublinear memory. *Theoretical Computer Science*, 849:22–34, 2021. doi:10.1016/J.TCS.2020.10.007.

13 Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The complexity of $(\delta+1)$ coloring in congested clique, massively parallel computation, and centralized local computation. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 471–480, 2019. doi:10.1145/3293611.3331607.

14 Artur Czumaj, Jakub Łącki, Aleksander Mądry, Slobodan Mitrović, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. In *ACM Symposium on Theory of Computing (STOC)*, pages 471–484, 2018.

15 Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2017. doi:10.1145/3087556.3087580.

16 Guy Even, Mohsen Ghaffari, and Moti Medina. Distributed set cover approximation: primal-dual with optimal locality. In *International Symposium on Distributed Computing (DISC)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

17 Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrović, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for MIS, matching, and vertex cover. In *ACM Symposium on Principles of Distributed Computing*, pages 129–138, 2018. doi:10.1145/3212734.3212743.

18 Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrović. Improved parallel algorithms for density-based network clustering. In *International Conference on Machine Learning (ICML)*, pages 2201–2210. PMLR, 2019. URL: http://proceedings.mlr.press/v97/ghaffari19a.html.

19 Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1636–1653. SIAM, 2019. doi:10.1137/1.9781611975482.99.

20 Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *International Symposium on Algorithms and Computation*, pages 374–383. Springer, 2011. doi:10.1007/978-3-642-25591-5_39.

21 Oussama Hanguir and Clifford Stein. Distributed algorithms for matching in hypergraphs. In *Workshop on Approximation and Online Algorithms (WAOA)*, pages 30–46. Springer, 2021.

22 Nicholas JA Harvey, Christopher Liaw, and Paul Liu. Greedy and local ratio algorithms in the mapreduce model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 43–52, 2018.

23 Dorit S Hochbaum. Approximation algorithms for the set covering and vertex cover problems. *SIAM Journal on Computing*, 11(3):555–556, 1982. doi:10.1137/0211045.

24 Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley Longman Publishing Co., Inc., USA, 1992.

25 Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948. SIAM, 2010. doi:10.1137/1.9781611973075.76.

26 Richard M Karp and Vijaya Ramachandran. A survey of parallel algorithms for shared-memory machines, 1988.

27 Samir Khuller, Uzi Vishkin, and Neal Young. A primal-dual parallel approximation technique applied to weighted set and vertex covers. *Journal of Algorithms*, 17(2):280–289, 1994. doi:10.1006/JAGM.1994.1036.

28 Christos Koufogiannakis and Neal E Young. Distributed algorithms for covering, packing and maximum weighted matching. *Distributed Computing*, 24:45–63, 2011. doi:10.1007/S00446-011-0127-7.

29 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2006.

30 Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 85–94, 2011. doi:10.1145/1989493.1989505.

31 Christoph Lenzen and Roger Wattenhofer. Brief announcement: Exponential speed-up of local algorithms using non-local communication. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 295–296, 2010. doi:10.1145/1835698.1835772.

32 Paul Liu and Jan Vondrak. Submodular optimization in the mapreduce model. In *2nd Symposium on Simplicity in Algorithms (SOSA 2019)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2019.

33 Alessandro Panconesi and Aravind Srinivasan. On the complexity of distributed network decomposition. *Journal of Algorithms*, 20(2):356–374, 1996. doi:10.1006/JAGM.1996.0017.

34 Sridhar Rajagopalan and Vijay V Vazirani. Primal-dual RNC approximation algorithms for set cover and covering integer programs. *SIAM Journal on Computing*, 28(2):525–540, 1998. doi:10.1137/S0097539793260763.

35 Sanguthevar Rajasekaran and John H Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, 1989. doi:10.1137/0218041.

36 Stergios Stergiou and Kostas Tsioutsiouliklis. Set cover at web scale. In *ACM International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 1125–1133, 2015. doi:10.1145/2783258.2783315.

37 Alastair J Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, 8(10):127–128, 1974.

38 David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms.* Cambridge University Press, USA, 1st edition, 2011.

A Analysis of the Set Sampling Process

In this section we prove two properties of the SSP, which are key in analyzing our algorithms. Recall the set sampling process SSP: initially $A_k = A$, and R_k is obtained by including each element of A_k independently with probability p_k. Then, for every i from $k-1$ down to 0, an adversary chooses $A_i \subseteq A_{i+1}$ (possibly with randomization) and then R_i is obtained by including every element of A_i independently with probability p_i.

Note two things about this process. First, the adversary can be randomized. Second, the adversary can be *adaptive*: its choice of A_i can depend on A_j and R_j for $j > i$. Recall that after running this process, z is the largest index such that R_z is nonempty. Our goal in this section is to prove the following lemma:

▶ **Lemma 3.** *Consider the SSP using any adversary and $\epsilon > 0$. Then, $\mathbb{E}[|R_z|] \leq 1 + 4\epsilon$.*

To get some intuition for why Lemma 3 might be true, observe that the sampling probability p_i increases very slowly; specifically, it increases by a $(1+\epsilon)$ factor every b steps. So the algorithm gets many chances at each (low) probability to obtain a non-empty R_i, and so it is not very likely to get more than 1 element in R_z.

To prove this lemma, we start with a simple but extremely useful observation: we may assume without loss of generality that the adversary is *nonadaptive*: its choice of A_i does not depend on R_j for $j > i$ (it can still depend on A_j for $j > i$). In other words, a nonadaptive adversary must pick the entire sequence of A_i's before seeing the results of any of the R_i's. Moreover, we may assume that the adversary is *deterministic*.

▶ **Observation 11.** *Without loss of generality, the adversary is nonadaptive and deterministic, i.e., it is a single fixed sequence $A_k, A_{k-1}, \dots, A_0$.*

Proof. We begin by showing that the adversary is nonadaptive without loss of generality. To see this, suppose there is some adaptive adversary P. Then let P' be the nonadaptive adversary obtained by simply running P under the assumption that every $R_i = \emptyset$. Clearly, this gives a (possibly randomized) sequence $A_k, A_{k-1}, \dots, A_0$ without needing to see the R_i's, and so is nonadaptive. Clearly, P and P' behave identically until $z-1$, i.e., until just *after* the first time that some R_i is nonempty (since P' is just P under the assumption that all R_i's are empty). But indices $z-1$ down to 0 make no difference in Lemma 3! Hence if Lemma 3 holds for nonadaptive adversaries, it also holds for adaptive adversaries.

So we assume that the adversary is nonadaptive, i.e., the adversarial choice is simply a distribution over sequences $A_k, A_{k-1}, \dots, A_0$. This means that the expectation in Lemma 3 is taken over both the adversary's random choices and the randomness from sampling the R_i's once the A_i's are fixed. These are intermixed for an adaptive adversary but for a nonadaptive adversary, which we may assume WLOG, we can separate these out by first choosing the random A_i's and then subsampling to get the R_i's. So we want to prove that

$$\mathbb{E}_{A_k,\dots,A_0}\left[\mathbb{E}_{R_k,\dots R_0}[|R_z|]\right] \leq 1 + 4\epsilon.$$

Suppose we could prove Lemma 3 for a deterministic nonadaptive adversary, i.e., for a fixed $A_k, A_{k-1}, \dots, A_0$. In other words, suppose that $\mathbb{E}_{R_k,\dots,R_0}|R_z| \leq 1 + 4\epsilon$ for all sequences $A_k, A_{k-1}, \dots, A_0$. Then clearly

$$\mathbb{E}_{A_k,\dots,A_0}\left[\mathbb{E}_{R_k,\dots R_0}[|R_z|]\right] \leq \mathbb{E}_{A_k,\dots,A_0}[1 + 4\epsilon] = 1 + 4\epsilon.$$

Thus if we can prove Lemma 3 against a nonadaptive deterministic adversary, we have proved Lemma 3 against an adaptive and randomized adversary, as desired. ◀

So from now on, we may assume that the family $A_k, A_{k-1}, \ldots, A_0$ is **fixed**. Note that in this setting, the sets R_i are independent of each other; this holds as the sets A_i are fixed, and the randomness used to obtain R_i is independent of the randomness used to sample other R_j sets. Before proving Lemma 3, we first show several auxiliary observations (all of which are in the setting where $A_k, A_{k-1}, \ldots, A_0$ are fixed).

Our first observation is that for the first b rounds of the SSP, the expected number of sampled elements is small.

▶ **Observation 12.** *Assume that $k \geq b\lceil\log_{1+\epsilon}(n_k/\epsilon)\rceil$. Then, for each $j \in (k-b, k]$, $p_j \cdot n_j \leq \epsilon$.*

Proof. We have

$$\lceil j/b \rceil = \lceil k/b \rceil \geq \log_{1+\epsilon}(n_k/\epsilon),$$

which gives

$$p_j \cdot n_j = (1+\epsilon)^{-\lceil j/b \rceil} \cdot n_j \leq (1+\epsilon)^{-\log_{1+\epsilon}(n_k/\epsilon)} \cdot n_k = \epsilon. \qquad ◀$$

We can also show that probabilities and the expected number of sampled elements do not increase much in any consecutive b steps.

▶ **Observation 13.** *For each $i \in [0, k)$, and any $j \in [i+1, i+b]$, $p_i \leq (1+\epsilon)p_j$ and $p_i \cdot n_i \leq (1+\epsilon)p_j \cdot n_j$.*

Proof. This first claim follows from the definition of p_i. The second claim additionally uses the fact that $n_0, \ldots, n_k$ is a non-decreasing sequence. ◀

This observation now allows us to show that if we have a relatively large expected number of elements in R_i, then the probability that we have not yet sampled any elements in R_j for $j > i$ is notably smaller than the probability that we haven't sampled any elements in R_j for $j > i + b$.

▶ **Lemma 14.** *Assume that $p_i \cdot n_i \geq \epsilon(1+\epsilon)$ for some $i \in [0, k-b]$. Then, $P(z \leq i) \leq P(z \leq i+b)/(2+2\epsilon)$.*

Proof. Denote by E_j the event that $R_j = \emptyset$. Recall that z is the maximum index such that $R_z \neq \emptyset$. Observe that the event that $z \leq x$ is equivalent to $\bigcap_{j>x} E_j$ and the individual events E_j are independent. Hence $P(z \leq i) = P(z \leq i+b) \cdot \prod_{j=i+1}^{i+b} P(E_j)$. To complete the proof we will show that $\prod_{j=i+1}^{i+b} P(E_j) \leq 1/(2+2\epsilon)$.

By Observation 13, we have that for $j \in [i+1, \ldots, b]$, $(1+\epsilon)p_j \cdot n_j \geq p_i \cdot n_i \geq \epsilon(1+\epsilon)$, which implies $p_j \cdot n_j \geq \epsilon$. Hence,

$$P(E_j) = (1-p_j)^{n_j} \leq e^{-p_j \cdot n_j} \leq e^{-\epsilon}.$$

By using the above, we get

$$\prod_{j=i+1}^{i+b} P(E_j) \leq e^{-b \cdot \epsilon} = e^{-\lceil \log(2(1+\epsilon))/\epsilon \rceil \epsilon} \leq e^{-\log(2(1+\epsilon))} = \frac{1}{2+2\epsilon}$$

which finishes the proof. ◀

This now allows us to give an absolute bound on the probability that we have not sampled any elements before we sample R_j, assuming that we have a pretty high probability of sampling an element in R_j.

▶ **Lemma 15.** *Let $j \in [0, k]$ be such that $p_j \cdot n_j \geq \epsilon(1+\epsilon)^c$ for a nonnegative integer c. Then $P(z \leq j) \leq (2+2\epsilon)^{-c}$.*

Proof. We prove the claim using induction on c. For $c = 0$, $(2+2\epsilon)^{-c}$ is 1, and the claim is trivially true.

Now, fix $c \geq 1$. By Observation 12, we have that $j \leq k - b$, and so we apply Lemma 14 to obtain $P(z \leq j) \leq P(z \leq j+b)/(2+2\epsilon)$. Hence, to complete the proof, it suffices to show $P(z \leq j+b) \leq (2+2\epsilon)^{-c+1}$.

We achieve that by applying the induction hypothesis to $j' = j + b$. Indeed, by Observation 13, $p_{j'} \cdot n_{j'} \geq \epsilon(1+\epsilon)^{c-1}$, and so the assumptions of the inductive hypothesis hold. As a result, we obtain $P(z \leq j+b) = P(z \leq j') \leq (2+2\epsilon)^{-c+1}$, as required. ◀

Before finally proving Lemma 3, we first show two more useful claims.

▷ Claim 16. For each $i \in [0, k]$, it holds that $\mathbb{E}[|R_i| \mid R_i \neq \emptyset] \leq 1 + p_i \cdot n_i$.

Proof. We have

$$\mathbb{E}[|R_i| \mid R_i \neq \emptyset] = \frac{\mathbb{E}[|R_i|]}{P(R_i \neq \emptyset)} = \frac{p_i \cdot n_i}{1-(1-p_i)^{n_i}} \leq \frac{p_i \cdot n_i}{1-\frac{1}{e^{p_i \cdot n_i}}} \leq \frac{p_i \cdot n_i}{1-\frac{1}{1+p_i \cdot n_i}} = \frac{p_i \cdot n_i}{\frac{p_i \cdot n_i}{1+p_i \cdot n_i}} = 1+p_i \cdot n_i.$$

Note that in the first inequality we used the fact that $1 - p_i \leq e^{-p_i}$, while in the second we used $e^{p_i \cdot n_i} \geq 1 + p_i \cdot n_i$. ◁

▷ Claim 17. Recall that z is the maximum index such that $R_z \neq \emptyset$. It holds that $\mathbb{E}[|R_z|] \leq 1 + \mathbb{E}[p_z \cdot n_z]$.

Proof. Denote by X_i the event that $R_j = \emptyset$ *for all* $j \in [i+1, k]$. Note that $z = i$ is the intersection of events $R_i \neq \emptyset$ and X_i.

$$\begin{aligned}
\mathbb{E}[|R_z|] &= \sum_{i=0}^{k} \mathbb{E}[|R_i| \mid z = i] \cdot P(z = i) \\
&= \sum_{i=0}^{k} \mathbb{E}[|R_i| \mid R_i \neq \emptyset \cap X_i] \cdot P(z = i) = \sum_{i=0}^{k} \mathbb{E}[|R_i| \mid R_i \neq \emptyset] \cdot P(z = i) \\
&\leq \sum_{i=0}^{k} (1 + p_i \cdot n_i) \cdot P(z = i) = \sum_{i=0}^{k} P(z = i) + \sum_{i=0}^{k} p_i \cdot n_i \cdot P(z = i) = 1 + \mathbb{E}[p_z \cdot n_z],
\end{aligned}$$

where the final inequality is from Claim 16. We used the fact that $\mathbb{E}[|R_i| \mid R_i \neq \emptyset \cap X_i] = \mathbb{E}[|R_i| \mid R_i \neq \emptyset]$, which follows from the fact that R_i is independent from X_i. ◁

We are now ready to prove Lemma 3.

Proof of Lemma 3. We know from Observation 11 that without loss of generality, the sequence $A_k, A_{k-1}, \ldots, A_0$ is fixed. By Claim 17 it suffices to show that $\mathbb{E}[p_z \cdot n_z] \leq 4\epsilon$. Let us define $X := p_z \cdot n_z$ to shorten notation.

$$\begin{aligned}
\mathbb{E}[X] &\leq P(X < \epsilon) \cdot \epsilon + \sum_{c=0}^{\infty} P\left(X \in \left[\epsilon(1+\epsilon)^c, \epsilon(1+\epsilon)^{c+1}\right)\right) \cdot \epsilon(1+\epsilon)^{c+1} \\
&\leq \epsilon + \sum_{c=0}^{\infty} P\left(X \geq \epsilon(1+\epsilon)^c\right) \cdot \epsilon(1+\epsilon)^{c+1} \\
&\leq \epsilon + \sum_{c=0}^{\infty} 2^{-c}(1+\epsilon)^{-c} \cdot \epsilon(1+\epsilon)^{c+1} \\
&\leq \epsilon(1 + 2(1+\epsilon)) \leq 4\epsilon.
\end{aligned}$$

Note that we used the bound on $P(X \geq \epsilon(1+\epsilon)^c) \leq (2+2\epsilon)^{-c}$ which follows directly from Lemma 15. ◀

A.1 Probability of the Sampled Element Being Not Unique

We use the following lemma to analyze our HypergraphMatching algorithm. Specifically, it upper bounds the probability that an edge is sampled in Algorithm 4, but *not* included in the final matching. In the proof of Lemma 10, we specify how to map our HypergraphMatching algorithm to the setup in this section.

▶ **Lemma 5.** *Let $a \in A_k$ and let $\epsilon \leq 1/2$. Then $P(|R_z| > 1 \mid a \in R_z) \leq 6\epsilon$.*

Proof. As for the previous proof in this section, first assume that the sets $A_0, \dots, A_k$ are fixed.

Our goal is to upper bound

$$P(|R_z| > 1 \mid a \in R_z) = \frac{P(|R_z| > 1 \cap a \in R_z)}{P(a \in R_z)} \tag{3}$$

Let $m_a = \min\{i \in [0,k] \mid a \in A_i\}$ be the index of the last step before a is removed from the sets $A_0, \dots, A_k$. We obtain:

$$\begin{aligned} P(|R_z| > 1 \cap a \in R_z) &= \sum_{i=m_a}^{k} P(z = i \cap |R_i| > 1 \cap a \in R_i) \\ &= \sum_{i=m_a}^{k} P(z \leq i \cap |R_i| > 1 \cap a \in R_i) && \text{since } a \in R_i \text{ implies } z \geq i \\ &= \sum_{i=m_a}^{k} P(z \leq i) P(|R_i| > 1 \cap a \in R_i) && \substack{z \leq i \text{ is equivalent to } R_j = \emptyset \text{ for all } j > i \\ \text{these events are independent of } R_i} \end{aligned}$$

Observe that the event $|R_i| > 1 \cap a \in R_i$ happens when a is sampled and at least one out of the remaining $n_i - 1$ elements of A_i are sampled. Hence,

$$P(|R_i| > 1 \cap a \in R_i) = p_i \cdot (1 - (1-p_i)^{n_i - 1}) \leq p_i \cdot (1 - (1 - p_i \cdot (n_i - 1))) \leq p_i^2 \cdot n_i,$$

and so we finally obtain $P(|R_z| > 1 \cap a \in R_z) \leq \sum_{i=m_a}^{k} P(z \leq i) p_i^2 \cdot n_i$. For the first inequality above, we used Bernoulli's inequality which states that $(1+x)^r \geq 1 + rx$ for every integer $r \geq 1$ and a real number $x \geq -1$. Analogous reasoning allows us to show a similar identity for the denominator:

$$P(a \in R_z) = \sum_{i=m_a}^{k} P(z \leq i) P(a \in R_i) = \sum_{i=m_a}^{k} P(z \leq i) p_i$$

Hence we can upper bound Equation (3) as follows

$$P(|R_z| > 1 \mid a \in R_z) \leq \frac{\sum_{i=m_a}^{k} P(z \leq i) P(a \in A_i) p_i^2 \cdot n_i}{\sum_{i=m_a}^{k} P(z \leq i) P(a \in A_i) p_i}. \tag{4}$$

▷ Claim 18. Let $I = \{i \in [m_a, k] \mid p_i \cdot n_i < \epsilon(1+\epsilon)\}$ be a set of indices. Then $\sum_{i=m_a}^{k} P(z \leq i) p_i^2 \cdot n_i \leq 2/(1-\epsilon) \sum_{i \in I} P(z \leq i) p_i^2 \cdot n_i$

Proof. Consider the sum

$$\sum_{i=m_a}^{k} P(z \leq i)p_i^2 \cdot n_i.$$

We are going to charge the summands indexed by $[m_a, k] \setminus I$ to the summands indexed by I. Formally, the charging is defined by a function $f : [m_a, k] \to [m_a, k]$. We define $f(i)$ to be the smallest index $j \in \{i, i+b, i+2b, \ldots\}$ such that $j \in I$, that is $p_j \cdot n_j < \epsilon(1+\epsilon)$. We note that $f(i)$ is well-defined since by Observation 12 we have that $(k-b, k] \subseteq I$.

Now we charge each summand i to $f(i) \in I$ and show that the total charge of each summand in I increases by at most a constant factor. Let us now fix any $j \in I$ and consider the sum $\sum_{i \in f^{-1}(j)} P(z \leq i)p_i^2 \cdot n_i$. Let $h := |f^{-1}(j)|$. Then $f^{-1}(j) = \{j, j-b, \ldots, j-(h-1)b\}$. We now show that the summands in the considered sum are geometrically decreasing (if we consider the indices in decreasing order). Indeed, consider $x \in f^{-1}(j) \setminus \{j\}$. We are now going to use the following facts.

- By Lemma 14 we have $P(z \leq x) \leq P(z \leq x+b)/(2+2\epsilon)$.
- By Observation 13, $p_x \cdot n_x \leq (1+\epsilon)p_{x+b} \cdot n_{x+b}$.
- By Observation 13, $p_x \leq (1+\epsilon)p_{x+b}$.

These three facts together imply that for any $x \in f^{-1}(j) \setminus \{j\}$

$$P(z \leq x)p_x^2 \cdot n_x \leq (1+\epsilon)/2 \cdot P(z \leq x+b)p_{x+b}^2 \cdot n_{x+b}.$$

Hence, the summands in $f^{-1}(j)$ can be arranged into a sequence in which the largest element is the summand corresponding to j, and each subsequent summand is at least a factor of $(1+\epsilon)/2$ smaller. As a result, the total charge of the summand j is $1/(1-(1+\epsilon)/2) = 2/(1-\epsilon)$. ◁

Using the above claim, we upper bound Equation (4).

$$\frac{\sum_{i=m_a}^{k} P(z \leq i)p_i^2 \cdot n_i}{\sum_{i=m_a}^{k} P(z \leq i)p_i} \leq \frac{2 \cdot \sum_{i \in I} P(z \leq i)p_i^2 \cdot n_i}{(1-\epsilon) \cdot \sum_{i \in I} P(z \leq i)p_i} < \frac{2 \cdot \epsilon(1+\epsilon) \cdot \sum_{i \in I} P(z \leq i)p_i}{(1-\epsilon) \cdot \sum_{i \in I} P(z \leq i)p_i} \leq 6\epsilon.$$

The proofs are stated while assuming that the sets $A_0, \ldots, A_k$ are fixed. As given by Observation 11, this assumption can be made without loss of generality. ◀

The Computational Power of Discrete Chemical Reaction Networks with Bounded Executions

David Doty
Computer Science, University of California – Davis, CA, USA

Ben Heckmann
CIT, Technical University of Munich, Germany
Computer Science, University of California – Davis, CA, USA

Abstract

Chemical reaction networks (CRNs) model systems where molecules interact according to a finite set of *reactions* such as $A + B \to C$, representing that if a molecule of A and B collide, they disappear and a molecule of C is produced. CRNs can compute Boolean-valued predicates $\phi : \mathbb{N}^d \to \{0, 1\}$ and integer-valued functions $f : \mathbb{N}^d \to \mathbb{N}$; for instance $X_1 + X_2 \to Y$ computes the function $\min(x_1, x_2)$, since starting with x_i copies of X_i, eventually $\min(x_1, x_2)$ copies of Y are produced.

We study the computational power of *execution bounded* CRNs, in which only a finite number of reactions can occur from the initial configuration (e.g., ruling out reversible reactions such as $A \rightleftharpoons B$). The power and composability of such CRNs depend crucially on some other modeling choices that do not affect the computational power of CRNs with unbounded executions, namely whether an initial leader is present, and whether (for predicates) all species are required to "vote" for the Boolean output. If the CRN starts with an initial leader, and can allow only the leader to vote, then all semilinear predicates and functions can be stably computed in $O(n \log n)$ parallel time by execution bounded CRNs.

However, if no initial leader is allowed, all species vote, and the CRN is "non-collapsing" (does not shrink from initially large to final $O(1)$ size configurations), then execution bounded CRNs are severely limited, able to compute only *eventually constant* predicates. A key tool is a characterization of execution bounded CRNs as precisely those with a nonnegative *linear potential function* that is strictly decreased by every reaction [6].

2012 ACM Subject Classification Theory of computation → Models of computation

Keywords and phrases chemical reaction networks, population protocols, stable computation

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.20

Related Version *Full Version*: https://arxiv.org/abs/2405.08649

Funding *David Doty*: NSF awards 2211793, 1900931, 1844976, and DoE EXPRESS award SC0024467.
Ben Heckmann: NSF award 1844976.

1 Introduction

Chemical reaction networks (CRNs) are a fundamental tool for understanding and designing molecular systems. By abstracting chemical reactions into a set of finite, rule-based transformations, CRNs allow us to model the behavior of complex chemical systems. For instance, the CRN with a single reaction $2X \to Y$, produces one Y every time two X molecules randomly react together, effectively calculating the function $f(x) = \lfloor x/2 \rfloor$ if the initial count of X is interpreted as the input and the eventual count of Y as the output. A commonly studied special case of CRNs is the *population protocol* model of distributed computing [3], in which each reaction has exactly two reactants and two products, e.g., $A + B \to C + D$. This model assumes idealized conditions where reactions can proceed indefinitely, constrained only by the availability of reactants in the well-mixed solution.

Precisely the *semilinear* predicates $\phi : \mathbb{N}^d \to \{0,1\}$ [1] and functions $f : \mathbb{N}^d \to \mathbb{N}$ [5] can be computed *stably*, roughly meaning that the output is correct no matter the order in which reactions happen. In population protocols or other CRNs with a finite reachable configuration space, this means that the output is correct with probability 1 under a stochastic scheduler that picks the next molecules to react at random. However, existing constructions to compute semilinear predicates and functions use CRNs with *unbounded executions*, meaning that it is possible to execute infinitely many reactions from the initial configuration. CRNs with *bounded executions* have several advantages. With an absolute guarantee on how many reactions will happen before the CRN terminates, wet-lab implementations need only supply a bounded amount of fuel to power the reactions. Such CRNs are simpler to reason about: each reaction brings it "closer" to the answer. They also lead to a simpler definition of stable computation than is typically employed: an execution bounded CRN stably computes a predicate/function if it gets the correct answer after sufficiently many reactions.

To study this topic, we study networks that must eventually reach a configuration where no further reactions can occur, regardless of the sequence of reactions executed. This restriction is nontrivial because the techniques of [5, 7] rely on reversible reactions (leading to unbounded executions) catalyzed by species we expect to be depleted once a computational step has terminated. This trick seems to add computational power to our system by undoing certain reactions as long as a specific species is present. Consider the following CRN computing $f(x_1, x_2, x_3) = \min(x_1 - x_2, x_3)$. The input values x_i are given by the counts of X_i, and the output by the count of Z molecules in the stable state:

$$X_1 \to Y \tag{1}$$

$$X_2 + Y \to \varnothing \tag{2}$$

$$Y + X_3 \to Z \tag{3}$$

$$Z + X_2 \to X_2 + X_3 + Y \tag{4}$$

Reactions (1) and (2) compute $x_1 - x_2$, storing the result in the count of Y. Next, reaction (3) can be applied exactly $\min(y, x_3)$ times. But since the order of reactions is a stochastic process, we might consume copies of Y in (3), before all of x_2 is subtracted from it. Therefore, we add reaction (4), using X_2 as a catalyst to undo reaction (3) as long as copies of X_2 are present, indicating that the first step of computation has not terminated. However, this means the above CRN does not have bounded exectutions, since reactions (3) and (4) can be alternated in an infinite execution. A similar technique is used in [5], where semilinear sets are understood as a finite union of linear sets, shown to be computable in parallel by CRNs. A reversible, catalyzed reaction finally converts the output of one of the CRNs to the global output. Among other questions, we explore how the constructions of [5] and [7] can be modified to provide equal computational power while guaranteeing bounded execution.

The paper is organized as follows. Section 3 defines execution boundedness (Definition 3.1). We introduce alternative characterizations of the class for use in later proofs, such as the lack of self-covering execution paths. Section 4 and 5 contain the main positive results of the paper and provide the concrete constructions used to decide semilinear sets and functions using execution bounded CRNs whose initial configurations contain a single leader. Section 6 discusses the limitations of execution bounded CRNs, introducing the concept of a "linear potential function" as a core characterization of these systems. We demonstrate that entirely execution bounded CRNs that are leaderless and non-collapsing (such as all population protocols), can only stably decide trivial semilinear predicates: the *eventually constant* predicates (Definition 6.6).

2 Preliminaries

We use established notation from [5, 7] and stable computation definitions from [3] for (discrete) chemical reaction networks.

2.1 Notation

Let $\mathbb{N}$ denote the nonnegative integers. For any finite set Λ, we write $\mathbb{N}^\Lambda$ to mean the set of functions $f : \Lambda \to \mathbb{N}$. Equivalently, $\mathbb{N}^\Lambda$ can be interpreted as the set of vectors indexed by the elements of Λ, and so $\mathbf{c} \in \mathbb{N}^\Lambda$ specifies nonnegative integer counts for all elements of Λ. $\mathbf{c}(i)$ denotes the i-th coordinate of $\mathbf{c}$, and if $\mathbf{c}$ is indexed by elements of Λ, then $\mathbf{c}(Y)$ denotes the count of species $Y \in \Lambda$. We sometimes use multiset notation for such vectors, e.g., $\{A, 3C\}$ for the vector $(1, 0, 3)$, assuming there are three species A, B, C. If $\Sigma \subseteq \Lambda$, then $\mathbf{i} \upharpoonright \Sigma$ denotes restriction of $\mathbf{i}$ to Σ.

For two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^k$, we write $\mathbf{x} \geqq \mathbf{y}$ to denote that $\mathbf{x}(i) \geq \mathbf{y}(i)$ for all $1 \leq i \leq k$, $\mathbf{x} \geq \mathbf{y}$ to denote that $\mathbf{x} \geqq \mathbf{y}$ but $\mathbf{x} \neq \mathbf{y}$, and $\mathbf{x} > \mathbf{y}$ to denote that $\mathbf{x}(i) > \mathbf{y}(i)$ for all $1 \leq i \leq k$. In the case that $\mathbf{y} = \mathbf{0}$, we say that $\mathbf{x}$ is *nonnegative*, *semipositive*, and *positive*, respectively. Similarly define $\leqq, \leq, <$.

For a matrix or vector $\mathbf{x}$, define $\|\mathbf{x}\| = \|\mathbf{x}\|_1 = \sum_i |\mathbf{x}(i)|$, i ranges over all the entries of $\mathbf{x}$.

2.2 Chemical Reaction Networks

A *chemical reaction network* (CRN) is a pair $\mathcal{C} = (\Lambda, R)$, where Λ is a finite set of chemical *species*, and R is a finite set of reactions over Λ, where each *reaction* is a pair $(\mathbf{r}, \mathbf{p}) \in \mathbb{N}^\Lambda \times \mathbb{N}^\Lambda$ indicating the *reactants* $\mathbf{r}$ and *products* $\mathbf{p}$. A *population protocol* [1] is a CRN in which all reactions $(\mathbf{r}, \mathbf{p})$ obey $\|\mathbf{r}\| = \|\mathbf{p}\| = 2$. (Note that CRNs, including population protocols, do not assume any underlying "communication graph" and model a well-mixed system in which each equal-sized of molecules is as likely to collide and react as any other.) We write reactions such as $A + 2B \to A + 3C$ to represent the reaction $(\{A, 2B\}, \{A, 3C\})$. A *configuration* $\mathbf{c} \in \mathbb{N}^\Lambda$ of a CRN assigns integer counts to every species $S \in \Lambda$. When convenient, we use the notation $\{n_1 S_1, n_2 S_2, \ldots, n_k S_k\}$ to describe a configuration $\mathbf{c}$ with $n_i \in \mathbb{N}$ copies of species S_i, i.e., $\mathbf{c}(S_i) = n_i$, and any species that is not listed is assumed to have a zero count. If some configuration $\mathbf{c}$ is understood from context, for a species S, we write $\#S$ to denote $\mathbf{c}(S)$. A reaction $(\mathbf{r}, \mathbf{p})$ is said to be *applicable* in configuration $\mathbf{c}$ if $\mathbf{r} \leqq \mathbf{c}$. If the reaction $(\mathbf{r}, \mathbf{p})$ is applicable, applying it results in configuration $\mathbf{c}' = \mathbf{c} - \mathbf{r} + \mathbf{p}$, and we write $\mathbf{c} \to \mathbf{c}'$.

An *execution* $\mathcal{E}$ is a finite or infinite sequence of one or more configurations $\mathcal{E} = (\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots)$ such that, for all $i \in \{1, \ldots, |\mathcal{E}| - 1\}, \mathbf{c}_{i-1} \to \mathbf{c}_i$ and $\mathbf{c}_{i-1} \neq \mathbf{c}_i$. $\mathbf{x} \Rightarrow_P \mathbf{y}$ denotes that P is finite, starts at $\mathbf{x}$, and ends at $\mathbf{y}$. In this case we say $\mathbf{y}$ is *reachable* from $\mathbf{x}$. Let $\mathsf{reach}(\mathbf{x}) = \{\mathbf{y} \mid \mathbf{x} \Rightarrow \mathbf{y}\}$. Note that the reachability relation is *additive*: if $\mathbf{x} \Rightarrow \mathbf{y}$, then for all $\mathbf{c} \in \mathbb{N}^\Lambda$, $\mathbf{x} + \mathbf{c} \Rightarrow \mathbf{y} + \mathbf{c}$.

For a CRN $\mathcal{C} = (\Lambda, R)$ where $|\Lambda| = n$ and $|R| = m$, define the $n \times m$ *stoichiometric matrix* $\mathbf{M}$ of $\mathcal{C}$ as follows. The species are ordered $S_1, \ldots, S_n$, and the reactions are ordered $(\mathbf{r}_1, \mathbf{p}_1), \ldots, (\mathbf{r}_m, \mathbf{p}_m)$, and $\mathbf{M}_{ij} = \mathbf{p}_j(S_i) - \mathbf{r}_j(S_i)$. In other words, $\mathbf{M}_{ij}$ is the net amount of S_i produced when executing the j'th reaction. For instance, if the CRN has two reactions $S_1 \to S_2 + 2S_3$ and $3S_2 + S_3 \to S_1 + S_2 + S_3$, then $\mathbf{M} = \begin{pmatrix} -1 & 1 \\ 1 & -2 \\ 2 & 0 \end{pmatrix}$.

▶ Remark 2.1. Let $\mathbf{u} \in \mathbb{N}^R$. Then the vector $\mathbf{Mu} \in \mathbb{Z}^\Lambda$ represents the change in species counts that results from applying reactions by amounts described in $\mathbf{u}$. In the above example, if $\mathbf{u} = (2, 1)$, then $\mathbf{Mu} = (-1, 0, 4)$, meaning that executing the first reaction twice ($\mathbf{u}(1) = 2$) and the second reaction once ($\mathbf{u}(2) = 1$) causes S_1 to decrease by 1, S_2 to stay the same, and S_3 to increase by 4.

2.3 Stable computation with CRNs

To capture the result of computations done by a CRN, we generalize the definitions to include information about how to interpret the final configuration after letting the CRN run until the result cannot change anymore (characterized below as *stable computation*). Computation primarily involves two classes of functions: 1. evaluating predicates $\phi : \mathbb{N}^k \to \{0, 1\}$ to determine properties of the input, and 2. executing general functions that map an input configuration to an output, denoted as $f : \mathbb{N}^k \to \mathbb{N}$.

The definitions below reference *input species* $\Sigma \subseteq \Lambda$ and an *initial context* $\mathbf{s} \in \mathbb{N}^{\Lambda \setminus \Sigma}$. If $\mathbf{s} = \mathbf{0}$ we say that CRN is *leaderless.* The initial context may be any constant multiset of species, though in practice it tends to be a single "leader" molecule. Furthermore, other initial contexts such as $\{2A, 3B\}$ could be produced from a single leader L via a reaction $L \to 2A + 3B$, so we may assume without loss of generality that the initial context, if it is nonzero, is simply a single leader. In both cases, we say $\mathbf{i} \in \mathbb{N}^\Lambda$ is a *valid initial configuration* if $\mathbf{i} = \mathbf{s} + \mathbf{x}$, where $\mathbf{x}(S) = 0$ for all $S \in \Lambda \setminus \Sigma$; i.e., $\mathbf{i}$ is the initial context plus only input species.

A *chemical reaction decider* (CRD) is a tuple $\mathcal{D} = (\Lambda, R, \Sigma, \Upsilon_1, \Upsilon_0, \mathbf{s})$, where (Λ, R) is a CRN, $\Sigma \subseteq \Lambda$ is the set of *input species*, $\Upsilon_1 \subseteq \Lambda$ is the set of *yes voters*, and $\Upsilon_0 \subseteq \Lambda$ is the set of *no voters*, such that $\Upsilon_1 \cap \Upsilon_0 = \emptyset$, and $\mathbf{s} \in \mathbb{N}^{\Lambda \setminus \Sigma}$ is the *initial context.* If $\Upsilon_1 \cup \Upsilon_0 = \Lambda$, we say the CRD is *all-voting.* We define a global output partial function $\Phi : \mathbb{N}^\Lambda \dashrightarrow \{0, 1\}$ as follows. $\Phi(\mathbf{c})$ is undefined if either $\mathbf{c} = \mathbf{0}$, or if there exist $S_0 \in \Upsilon_0$ and $S_1 \in \Upsilon_1$ such that $\mathbf{c}(S_0) > 0$ and $\mathbf{c}(S_1) > 0$. In other words, we require a unanimous vote as our output. We say $\mathbf{c}$ is *stable* if, for all $\mathbf{c}'$ such that $\mathbf{c} \Rightarrow \mathbf{c}'$, $\Phi(\mathbf{c}) = \Phi(\mathbf{c}')$. We say a CRD $\mathcal{D}$ *stably decides* the predicate $\psi : \mathbb{N}^\Sigma \to \{0, 1\}$ if, for any valid initial configuration $\mathbf{i} \in \mathbb{N}^\Lambda$, letting $\mathbf{i}_0 = \mathbf{i} \upharpoonright \Sigma$, for all configurations $\mathbf{c} \in \mathbb{N}^\Lambda, \mathbf{i} \Rightarrow \mathbf{c}$ implies $\mathbf{c} \Rightarrow \mathbf{c}'$ such that $\mathbf{c}'$ is stable and $\Phi(\mathbf{c}') = \psi(\mathbf{i}_0)$. We associate to a predicate ψ the set $A = \psi^{-1}(1)$ of inputs on which ψ outputs 1, so we can equivalently say the CRD *stably decides* the set A.

A chemical reaction computer (*CRC*) is a tuple $\mathcal{C} = (\Lambda, R, \Sigma, Y, \mathbf{s})$, where (Λ, R) is a CRN, $\Sigma \subset \Lambda$ is the set of *input species*, $Y \in \Lambda \setminus \Sigma$ is the *output species*, and $\mathbf{s} \in \mathbb{N}^{\Lambda \setminus \Sigma}$ is the *initial context.* A configuration $\mathbf{o} \in \mathbb{N}^\Lambda$ is *stable* if, for every $\mathbf{c}$ such that $\mathbf{o} \Rightarrow \mathbf{c}, \mathbf{o}(Y) = \mathbf{c}(Y)$, i.e. the output can never change again. We say that $\mathcal{C}$ *stably computes* a function $f : \mathbb{N}^k \to \mathbb{N}$ if for any valid initial configuration $\mathbf{i} \in \mathbb{N}^\Sigma$ and any $\mathbf{c} \in \mathbb{N}^\Lambda, \mathbf{i} \Rightarrow \mathbf{c}$ implies $\mathbf{c} \Rightarrow \mathbf{o}$ such that $\mathbf{o}$ is stable and $f(\mathbf{i} \upharpoonright \Sigma) = \mathbf{o}(Y)$.

2.4 Time model

The following model of stochastic chemical kinetics is widely used in quantitative biology and other fields dealing with chemical reactions between species present in small counts [8]. It ascribes probabilities to execution sequences, and also defines the time of reactions, allowing us to study the computational complexity of the CRN computation in Sections 4 and 5. If the volume is defined to be the total number of molecules, then the time model is essentially equivalent to the notion of *parallel time* studied in population protocols [2]. In this paper, the rate constants of all reactions are 1, and we define the kinetic model with this assumption. A reaction is *unimolecular* if it has one reactant and *bimolecular* if it has two reactants. We use no higher-order reactions in this paper.

The kinetics of a CRN is described by a continuous-time Markov process as follows. Given a fixed volume $v > 0$, the *propensity* of a unimolecular reaction $\alpha : X \to \dots$ in configuration $\mathbf{c}$ is $\rho(\mathbf{c}, \alpha) = \mathbf{c}(X)$. The propensity of a bimolecular reaction $\alpha : X + Y \to \dots$, where $X \neq Y$, is $\rho(\mathbf{c}, \alpha) = \frac{\mathbf{c}(X)\mathbf{c}(Y)}{v}$. The propensity of a bimolecular reaction $\alpha : X + X \to \dots$ is $\rho(\mathbf{c}, \alpha) = \frac{1}{2}\frac{\mathbf{c}(X)(\mathbf{c}(X)-1)}{v}$. The propensity function determines the evolution of the system as follows. The time until the next reaction occurs is an exponential random variable with rate $\rho(\mathbf{c}) = \sum_{\alpha \in R} \rho(\mathbf{c}, \alpha)$ (note that $\rho(\mathbf{c}) = 0$ if no reactions are applicable to $\mathbf{c}$). The probability that next reaction will be a particular α_{next} is $\frac{\rho(\mathbf{c},\alpha_{\text{next}})}{\rho(\mathbf{c})}$.

The kinetic model is based on the physical assumption of well-mixedness that is valid in a dilute solution. Thus, we assume the *finite density constraint*, which stipulates that a volume required to execute a CRN must be proportional to the maximum molecular count obtained during execution [12]. In other words, the total concentration (molecular count per volume) is bounded. This realistically constrains the speed of the computation achievable by CRNs.

For a CRD or CRC stably computing a predicate/function, the *stabilization time* is the function $t : \mathbb{N} \to \mathbb{N}$ defined for all $n \in \mathbb{N}$ as $t(n) =$ the worst-case expected time to reach from any valid initial configuration of size n to a stable configuration.

2.5 Semilinear sets, predicates, functions

▶ **Definition 2.2.** *A set $L \subseteq \mathbb{N}^d$ is* linear *if there are vectors $\mathbf{b}, \mathbf{p}_1, \dots, \mathbf{p}_k$ such that $L = \{\mathbf{b} + n_1\mathbf{p}_1 + \dots + n_k\mathbf{p}_k \mid n_1, \dots, n_k \in \mathbb{N}\}$. A set is* semilinear *if it is a finite union of linear sets. A predicate $\phi : \mathbb{N}^d \to \{0, 1\}$ is* semilinear *if the set $\phi^{-1}(1)$ is semilinear. A function $f : \mathbb{N}^d \to \mathbb{N}$ is* semilinear *if its* graph *$\{(\mathbf{x}, y) \in \mathbb{N}^{d+1} \mid f(\mathbf{x}) = y\}$ is semilinear.*

The following is a known characterization of the computational power of CRNs [3, 4].

▶ **Theorem 2.3** ([3, 4])**.** *A predicate/function is stably computable by a CRD/CRC if and only if it is semilinear.*

▶ **Definition 2.4.** *$T \subseteq \mathbb{N}^d$ is a* threshold *set is if there are constants $c, w_1, \dots, w_d \in \mathbb{Z}$ such that $T = \{\mathbf{x} \in \mathbb{N}^d \mid w_1\mathbf{x}(1) + \dots + w_d\mathbf{x}(d) \leq c\}$. $M \subseteq \mathbb{N}^d$ is a* mod *set if there are constants $c, m, w_1, \dots, w_d \in \mathbb{N}$ such that $M = \{\mathbf{x} \in \mathbb{N}^d \mid w_1\mathbf{x}(1) + \dots + w_d\mathbf{x}(d) \equiv c \mod m\}$.*

The following well-known characterization of semilinear sets is useful.

▶ **Theorem 2.5** ([9])**.** *A set is* semilinear *if and only if it is a Boolean combination (union, intersection, complement) of threshold and mod sets.*

3 Execution bounded chemical reaction networks

In this section, we define execution bounded CRNs and state an alternate characterization of the definition.

▶ **Definition 3.1.** *A CRN $\mathcal{C}$ is* execution bounded *from configuration $\mathbf{x}$ if all executions $\mathcal{E} = (\mathbf{x}, \dots)$ starting at $\mathbf{x}$ are finite. A CRD or CRC $\mathcal{C}$ is* execution bounded *if it is execution bounded from every valid initial configuration. $\mathcal{C}$ is* entirely execution bounded *if it is execution bounded from every configuration.*

This is a distinct concept from the notion of "bounded" CRNs studied by Rackoff [11] (studied under the equivalent formalism of vector addition systems). That paper defines a CRN to be *bounded* from a configuration $\mathbf{x}$ if $|\mathsf{reach}(\mathbf{x})|$ is finite (and shows that the decision problem of determining whether this is true is EXPSPACE-complete.) We use the term *execution bounded* to avoid confusion with this concept.

We first observe an equivalent characterization of execution bounded that will be useful in the negative results of Section 6.

▶ **Definition 3.2.** *A execution $\mathcal{E} = (\mathbf{x}_1, \mathbf{x}_2, \dots)$ is* self-covering *if for some $i < j$, $\mathbf{x}_i \leqq \mathbf{x}_j$. It is* strictly self-covering *if $\mathbf{x}_i \leq \mathbf{x}_j$. We also refer to these as (strict) self-covering* paths.[1]

▶ **Lemma 3.3.** *A CRN is execution bounded from $\mathbf{x}$ if and only if there is no self-covering path from $\mathbf{x}$.*

4 Execution bounded CRDs stably decide all semilinear sets

In this section, we will show that execution bounded CRDs have the same computational power as unrestricted CRDs. The following is the main result of this section.

▶ **Theorem 4.1.** *Exactly the semilinear sets are stably decidable by execution bounded CRDs. Furthermore, each can be stably decided with expected stabilization time $\Theta(n \log n)$.*

Since semilinear sets are Boolean combinations of mod and threshold predicates, we prove this theorem by showing that execution bounded CRDs can decide mod and threshold sets individually as well as any Boolean combination in the following lemmas. To ensure execution boundedness in the last step, we require the following property.

▶ **Definition 4.2.** *Let $\mathcal{D}$ be a CRD with voting species Υ. We say $\mathcal{D}$ is* single-voting *if for any valid initial configuration $\mathbf{i} \in \mathbb{N}^\Sigma$ and any $\mathbf{c} \in \mathbb{N}^\Lambda$ s.t. $\mathbf{i} \Rightarrow \mathbf{c}$, $\sum_{V \in \Upsilon} \mathbf{c}(V) = 1$, i.e., exactly one voter is present in every reachable configuration.*

Lemmas 4.3 and 4.4 are proven in the full version of this paper.

▶ **Lemma 4.3.** *Every mod set $M = \left\{ (x_1, \dots, x_d) \mid \sum_{i=1}^d w_i x_i \equiv c \bmod m \right\}$ is stably decidable by an execution bounded, single-voting CRD with expected stabilization time $\Theta(n \log n)$.*

We design a CRD $\mathcal{D}$ with exactly one leader present at all times, cycling through m "states" while consuming the input and accepting on state c. Let $\Sigma = \{X_1, \dots, X_d\}$ be the set of input species and start with only one L_0 leader, i.e. set the initial context $\mathbf{s}(L_0) = 1$ and $\mathbf{s}(S) = 0$ for all other species. For each $i \in \{1, \dots, d\}, j \in \{0, \dots, m-1\}$ add the following reaction: $X_i + L_j \to L_{j+w_i \bmod m}$. Let only L_c vote *yes* and all other species *no*, i.e. $\Upsilon = \{L_c\}$. For any valid initial configuration, $\mathcal{D}$ reaches a stable configuration which votes *yes* if and only if the input is in the mod set, and *no* otherwise.

▶ **Lemma 4.4.** *Every threshold set $T = \left\{(x_1, \dots, x_d) \mid \sum_{i=1}^d w_i x_i \geq t\right\}$ is stably decidable by an execution bounded, single-voting CRD with expected stabilization time $\Theta(n \log n)$.*

We design a CRD $\mathcal{D}$ which multiplies the input molecules according to their weight and consumes positive and negative units alternatingly using a single leader. Once no more reaction is applicable, the leader's state will indicate whether or not there are positive units left and the threshold is met. Let $\Sigma = \{X_1, \dots, X_d\}$ be the set of input species and $\Upsilon = \{L_Y\}$

[1] Rackoff [11] uses the term "self-covering" to mean what we call *strictly self-covering* here, and points out that Karp and Miller [10] showed that $|\mathsf{reach}(\mathbf{x})|$ is infinite if and only if there is a strictly self-covering path from $\mathbf{x}$. The distinction between these concepts is illustrated by the CRN $A \rightleftharpoons B$. From any configuration $\mathbf{x}$, $\mathsf{reach}(\mathbf{x})$ is finite ($|\mathsf{reach}(\mathbf{x})| = \mathbf{x}(A) + \mathbf{x}(B) + 1$), and there is no strict self-covering path. However, from (say) $\{A\}$, there is a (nonstrict) self-covering path $\{A\} \Rightarrow \{B\} \Rightarrow \{A\}$, and by repeating, this CRN has an infinite cycling execution within its finite configuration space $\mathsf{reach}(\{A\}) = \{\{A\}, \{B\}\}$.

the *yes* voter. We first add reactions to multiply the input species by their respective weights. For all $i \in \{1, \dots, d\}$, add the reaction:

$$X_i \to \begin{cases} w_i P & \text{if } w_i > 0 \\ -w_i N & \text{if } w_i < 0 \\ \emptyset & \text{otherwise} \end{cases} \tag{5}$$

P and N represent "positive" and "negative" units respectively. Now add reactions to consume P and N alternatingly using a leader until we run out of one species:

$$L_Y + N \to L_N \tag{6}$$

$$L_N + P \to L_Y \tag{7}$$

Finally, initialize the CRD with one L_Y and the threshold number t copies of P (or $-tN$ if t is negative), i.e. $\mathbf{s}(L_Y) = 1$, $\mathbf{s}(P) = t$ if $t > 0$, or $\mathbf{s}(N) = -t$ if $t < 0$, and $\mathbf{s}(S) = 0$ for all other species. For any valid initial configuration, $\mathcal{D}$ reaches a stable configuration which votes *yes* if and only if the weighted sum of inputs is above the threshold, and *no* otherwise.

▶ **Lemma 4.5.** *If sets $X_1, X_2 \subseteq \mathbb{N}^d$ are stably decided by some execution bounded, single-voting CRD, then so are $X_1 \cup X_2$, $X_1 \cap X_2$, and $\overline{X_1}$ with expected stabilization time $O(n \log n)$.*

Proof. To stably decide $\overline{X_1}$, swap the yes and no voters.

For $\cup$ and $\cap$, consider a construction where we decide both sets separately and record both of their votes in a new voter species. For this, we allow the set of all voters to be a strict subset of all species. We first add reactions to duplicate our input with reactions of the form

$$X_i \to X_{i,1} + X_{i,2} \tag{8}$$

by two separate CRDs. Subsequently, we add reactions to record the separate votes in one of four new voter species: $V_{NN}, V_{NY}, V_{YN}, V_{YY}$. The first and second CRN determine the first and second subscript respectively. For $b \in \{Y, N\}$ and if S_b, T_b are voters of $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively, add the reactions:

$$S_b + V_{\bar{b}?} \to S_b + V_{b?} \tag{9}$$

$$T_b + V_{?\bar{b}} \to T_b + V_{?b} \tag{10}$$

Above, the ? subscript is shorthand for "any bit"; e.g. if N_1 is the *no* voter of the first CRD, we would add two reactions $N_1 + L_{YN} \to N_1 + L_{NN}$ and $N_1 + L_{YY} \to N_1 + L_{NY}$. We let the *yes* voters be: $\Upsilon = \{V_{NY}, V_{YN}, V_{YY}\}$ to stably decide $X_1 \cup X_2$ or $\Upsilon = \{V_{YY}\}$ to stably decide $X_1 \cap X_2$.

Reaction (8) will complete in $O(\log n)$ time and is clearly execution bounded since the input X_i is finite and not produced in any reaction. Consequently, two separate CRNs run in $\Theta(n \log n)$ time as shown in Lemma 4.3 and Lemma 4.4. After stabilization of the parallel CRNs, we expect reaction (9) and (10) to happen exactly once. Each molecule involved is a leader and has count 1 in volume n. This leads to a rate of $\lambda = \frac{1 \cdot 1}{n}$, so the expected time for one reaction to happen is $O(n)$. It is important to note that reactions (9) and (10) do not result in unbounded executions due to the unanimous vote in parallel CRDs. In both mod sets and threshold sets, the leader changes its vote a maximum of $|\mathbf{i}|$ times, with only ever one leader present at any time. Again, we start with only one V_{bb} voter present initially and no reaction changes the count of voters, making our construction single-voting. ◀

Since semilinear predicates are exactly Boolean combinations of threshold and mod predicates, Lemmas 4.3–4.5 imply Theorem 4.1.

We can also prove the same result for all-voting CRDs. Note, however, that such CRDs cannot be "composed" using the constructions of Lemma 4.5 and Theorem 5.4, which crucially relied on the assumption that the CRDs being used as "subroutines" are single-voting.

▶ **Theorem 4.6.** *Every semilinear set is stably decidable by an execution bounded, all-voting CRD, with expected stabilization time $O(n \log n)$.*

5 Execution bounded CRCs stably compute all semilinear functions

In this section we shift focus from computing Boolean-valued predicates $\phi : \mathbb{N}^d \to \{0,1\}$ to integer-valued functions $f : \mathbb{N}^d \to \mathbb{N}$, showing that execution bounded CRCs can stably compute the same class of functions (semilinear) as unrestricted CRCs.

Similar to [5, 7], we compute semilinear functions by decomposing them into "affine pieces", which we will show can be computed by execution bounded CRNs and combined by using semilinear predicates to decide which linear function to apply for a given input.[2]

We say a partial function $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ is *affine* if there exist vectors $\mathbf{a} \in \mathbb{Q}^k$, $\mathbf{c} \in \mathbb{N}^k$ with $\mathbf{x} - \mathbf{c} \geq \mathbf{0}$ and nonnegative integer $b \in \mathbb{N}$ such that $f(\mathbf{x}) = \mathbf{a}^\top(\mathbf{x} - \mathbf{c}) + b$. For a partial function f we write $\operatorname{dom} f$ for the *domain* of f, the set of inputs for which f is defined. This definition of affine function may appear contrived, but the main utility of the definition is that it satisfies Lemma 5.3. For convenience, we can ensure to only work with integer valued molecule counts by multiplying by $\frac{1}{d}$ after the dot product, where d may be taken to be the least common multiple of the denominators of the rational coefficients in the original definition such that $n_i = d \cdot \mathbf{a}(i)$: $f(\mathbf{x}) = b + \sum_{i=1}^{k} \mathbf{a}(i)(\mathbf{x}(i) - \mathbf{c}(i)) \iff f(\mathbf{x}) = b + \frac{1}{d}\sum_{i=1}^{k} n_i(\mathbf{x}(i) - \mathbf{c}(i))$.

We say that a partial function $\hat{f} : \mathbb{N}^k \to \mathbb{N}^2$ is a *diff-representation* of f if $\operatorname{dom} f = \operatorname{dom} \hat{f}$ and, for all $\mathbf{x} \in \operatorname{dom} f$, if $(y_P, y_C) = \hat{f}(\mathbf{x})$, then $f(\mathbf{x}) = y_P - y_C$, and $y_P = O(f(\mathbf{x}))$. In other words, $\hat{f}$ represents f as the difference of its two outputs y_P and y_C, with the larger output y_P possibly being larger than the original function's output, but at most a multiplicative constant larger [7].

▶ **Lemma 5.1.** *Let $f : \mathbb{N}^k \to \mathbb{N}$ be an affine partial function. Then there is a diff-representation $\hat{f} : \mathbb{N}^k \longrightarrow \mathbb{N}^2$ of f and an execution bounded CRC that monotonically stably computes $\hat{f}$ in expected stabilization time $O(n)$.*

Proof. Define a CRC C with input species $\Sigma = \{X_1, \ldots, X_k\}$ and output species $\Gamma = \{Y^P, Y^C\}$. We need to ensure that after stabilizing, $y = \#Y^P - \#Y^C$

To account for the b offset, start with b copies of Y^P.

For the c_i offset, we must reduce the number of X_i by c_i. Since the result will be used in the next reaction, we want to produce a new species X_i' and require X_i' to not be consumed during the computation. We achieve this by adding reactions that let X_i consume itself c_i times (keeping track with a subscript) and converting X_i to X_i' once c_i has been reached. For the sake of notation below, assume input species X_i is actually named $X_{i,1}$. For each $i \in \{1, \ldots, k\}$ and $m, p \in \{1, \ldots, c_i\}$, if $m + p \leq c_i$, add the reaction

$$X_{i,m} + X_{i,p} \to X_{i,m+p} \tag{11}$$

[2] While this proof generalizes to multivariate output functions as in [5, 7], to simplify notation we focus on single output functions. Multi-valued functions $f : \mathbb{N}^d \to \mathbb{N}^l$ can be equivalently thought of as l separate single output functions $f_i : \mathbb{N}^d \to \mathbb{N}$, which can be computed in parallel by independent CRCs.

If $m+p>c_i$, add the reaction

$$X_{i,m}+X_{i,p}\to X_{i,c_i}+(m+p-c_i)\,X_i' \tag{12}$$

Runtime: In volume n, the rate of reactions (11) and (12) would be $\lambda \approx \frac{(x_i)^2}{n}$ (x_i molecules have the chance to react with any of the x_i-1 others), so the expected time for the next reaction is $\frac{n}{(x_i)^2}$. The expected time for the whole process is $\sum_{i=1}^{x_i}\frac{n}{i^2}=n\sum_{i=1}^{x_i}\frac{1}{i^2}=O(n)$. Further, the reactions are execution bounded since both strictly decrease the number of their reactants and exactly x_i-1 reactions will happen.

To account for the n_i/d coefficient, we multiply by n_i, then divide by d using similar reactions as for the subtraction. To multiply by n_i, add the following reaction for each $i\in\{1,\dots,k\}$:

$$X_i'\to\begin{cases} n_iD_1^P, & \text{if } n_i>0\\ (-n_i)\,D_1^C, & \text{if } n_i<0\end{cases} \tag{13}$$

For each $m,p\in\{1,\dots,d-1\}$, if $m+p\le d-1$, add the reactions

$$D_m^P+D_p^P\to D_{m+p}^P \tag{14}$$
$$D_m^C+D_p^C\to D_{m+p}^C \tag{15}$$

If $m+p\ge d$, add the reactions

$$D_m^P+D_p^P\to D_{m+p-d}^P+Y^P \tag{16}$$
$$D_m^C+D_p^C\to D_{m+p-d}^C+Y^C \tag{17}$$

Reactions (13) complete in expected time $O(\log n)$, while (16) and (17) complete in $O(n)$ by a similar analysis as for the first two reactions. As for execution boundedness, (13) is only applicable once for every X_i'; all other reactions start with a number of reactants which are a constant factor of X_i' and decrease the count of their reactants by one in each reaction. ◀

We require the following result due to Chen, Doty, Soloveichik [5], guaranteeing that any semilinear function can be built from affine partial functions.

▶ **Lemma 5.2** ([5]). *Let $f:\mathbb{N}^d\to\mathbb{N}$ be a semilinear function. Then there is a finite set $\{f_1:\mathbb{N}^d\to\mathbb{N},\dots,f_m:\mathbb{N}^d\to\mathbb{N}\}$ of affine partial functions, where each $\operatorname{dom} f_i$ is a linear set, such that, for each $\mathbf{x}\in\mathbb{N}^d$, if $f_i(\mathbf{x})$ is defined, then $f(\mathbf{x})=f_i(\mathbf{x})$, and $\bigcup_{i=1}^m \operatorname{dom} f_i=\mathbb{N}^d$.*

We strengthen Lemma 5.2 to show we may assume each dom f_i is disjoint from the others. This is needed not only to prove Theorem 5.4, but to correct the proof of Lemma 4.4 in [5], which implicitly assumed the domains are disjoint.

▶ **Lemma 5.3.** *Let $f:\mathbb{N}^d\to\mathbb{N}$ be a semilinear function. Then there is a finite set $\{f_1:\mathbb{N}^d\to\mathbb{N},\dots,f_m:\mathbb{N}^d\to\mathbb{N}\}$ of affine partial functions, where each $\operatorname{dom} f_i$ is a linear set, and $\operatorname{dom} f_i\cap\operatorname{dom} f_j=\emptyset$ for all $i\ne j$, such that, for each $\mathbf{x}\in\mathbb{N}^d$, if $f_i(\mathbf{x})$ is defined, then $f(\mathbf{x})=f_i(\mathbf{x})$, and $\bigcup_{i=1}^m \operatorname{dom} f_i=\mathbb{N}^d$.*

The next theorem shows that semilinear functions can be computed by execution bounded CRCs in expected time $O(n\log n)$.

▶ **Theorem 5.4.** *Let $f:\mathbb{N}^d\to\mathbb{N}$ be a semilinear function. Then there is an execution bounded CRC that stably computes f with expected stabilization time $O(n\log n)$.*

Proof. We employ the same construction of [5] with minor alterations. A CRC with input species $\Sigma = \{X_1, \ldots, X_d\}$ and output species $\Gamma = \{Y\}$. By Lemma 5.3, we decompose our semilinear function into partial affine functions (with linear, disjoint domains), which can be computed in parallel by Lemma 5.1. Further, we decide which function to use by computing the predicate $\phi_i =$ "x $\in$ dom f_i" (Theorem 4.1). We interpret each $\widehat{Y}_i^P$ and $\widehat{Y}_i^C$ as an "inactive" version of "active" output species Y_i^P and Y_i^C. Let L_i^Y, L_i^N be the *yes* and *no* voters respectively voting whether $\mathbf{x}$ lies in the domain of i-th partial function. Now, we convert the function result of the applicable partial affine function to the global output by adding the following reactions for each $i \in \{1, \ldots, m\}$.

$$L_i^Y + \widehat{Y}_i^P \to L_i^Y + Y_i^P + Y \tag{18}$$

$$L_i^N + Y_i^P \to L_i^N + M_i \tag{19}$$

$$M_i + Y \to \widehat{Y}_i^P \tag{20}$$

Reaction (18) produces an output copy of species Y and (19) and (20) reverse the first reaction using only bimolecular reactions. Both are catalyzed by the vote of the i-th predicate result. Also add reactions

$$L_i^Y + \widehat{Y}_i^C \to L_i^Y + Y_i^C \tag{21}$$

$$L_i^N + Y_i^C \to L_i^N + \widehat{Y}_i^C \tag{22}$$

and

$$Y_i^P + Y_i^C \to K \tag{23}$$

$$K + Y \to \varnothing \tag{24}$$

Reactions (21) and (22) activate and deactivate the "negative" output values and reactions (23) and (24) allow two active partial outputs to cancel out and consume the excess Y in the process. When the input is in the domain of function i, exactly one copy of L_i^Y will be present, otherwise one copy of L_i^N. Since we know that the predicate computation is execution bounded and produces at most one voter, the catalytic reaction will also happen at most as often as the leader changes its vote. Therefore, it is also execution bounded.

The underlying CRNs computing the predicates and functions have expected stabilization time $O(n \log n)$. Once they have stabilized, the slowest reactions described above are those where a leader (L_i^Y or L_i^N) must convert all outputs, which also takes expected time $O(n \log n)$ by a coupon collector argument. ◀

6 Limitations of execution bounded CRNs

The main positive results of the paper (Theorems 4.1 and 5.4) rely on the assumption that valid initial configurations have a single leader (in particular, they are execution bounded only from configurations with a single leader, but not from arbitrary configurations). Theorem 4.6 shows that we may assume the CRD deciding a semilinear set is all-voting. However, for the "constructive" results Lemma 4.5 and Theorem 5.4, which compose the output of a CRD $\mathcal{D}$ with downstream computation, using $\mathcal{D}$ as a "subroutine" to stably compute a more complex set/function, the constructions crucially use the assumption that $\mathcal{D}$ is single-voting (i.e., only the leader of $\mathcal{D}$ votes) to argue the resulting composed CRN is execution bounded. In this section we show these assumptions are necessary, proving that execution bounded CRNs without those constraints are severely limited in their computational abilities.

We use a result of Czerner, Guttenberg, Helfrich, and Esparza [6], showing that entirely execution bounded CRNs (from every configuration) can be characterized by a simpler property of having a "linear potential function" that essentially measures how close the CRN is to reaching a terminal configuration. We use this characterization to prove that entirely execution bounded CRNs can stably decide only limited semilinear predicates (eventually constant, Definition 6.6), assuming all species vote, and that molecular counts cannot decrease to $O(1)$ in stable configurations (see Definition 6.4).

6.1 Linear potential functions

We define a *linear potential function* of a CRN to be a nonnegative linear function of configurations that each reaction strictly decreases.

▶ **Definition 6.1.** *A* linear potential function $\Phi : \mathbb{R}^{\Lambda}_{\geq 0} \to \mathbb{R}_{\geq 0}$ *for a CRN is a nonnegative linear function, such that for each reaction* $(\mathbf{r}, \mathbf{p})$*,* $\Phi(\mathbf{p}) - \Phi(\mathbf{r}) < 0$.

Note that for a configuration $\mathbf{x}$, since $\Phi(\mathbf{x}) = \sum_{S \in \Lambda} v_S \mathbf{x}(S) \geq 0$, it must be nondecreasing in each species, i.e., all coefficients v_S must be nonnegative (though some are permitted to be 0). Intuitively, we can think of Φ as assigning a nonnegative "mass" to each species (the mass of S is v_S), such that each reaction removes a positive amount of mass from the system. Note also that since Φ is linear, the above is equivalent to requiring that $\Phi(\mathbf{p} - \mathbf{r}) < 0$, if we extend Φ to a linear function $\Phi : \mathbb{R}^{\Lambda} \to \mathbb{R}$ on vectors with negative elements.

A CRN may or may not have a linear potential function. Although it is not straightforward to "syntactically check" a CRN to see if has a linear potential function, it is efficiently decidable: a CRN has a linear potential function if and only if the following system of linear inequalities has a solution (which can be solved in polynomial time using linear programming techniques; the variables to solve for are the v_S for each $S \in \Lambda$), where the i'th reaction has reactants $\mathbf{r}_i$ and products $\mathbf{p}_i$, and species $S \in \Lambda$ has mass $v_S \geq 0$: $(\forall i) \sum_{S \in \Lambda} [\mathbf{p}_i(S) - \mathbf{r}_i(S)] v_S < 0$. For example, for the reactions $A + A \to B + C$ and $B + B \to A$, for each reaction to strictly decrease the potential function $\Phi(\mathbf{x}) = v_A \mathbf{x}(A) + v_B \mathbf{x}(B) + v_C \mathbf{x}(C)$, Φ must satisfy $2v_A > v_B + v_C$ and $2v_B > v_A$. In this case, $v_A = 1, v_B = 1, v_C = 0$ works.

▶ Remark 6.2. A system of linear inequalities with rational coefficients has a real solution if and only if it has a rational solution. For any homogeneous system (where all inequalities are comparing to 0), any positive scalar multiple of a solution is also a solution. By clearing denominators, a system has a rational solution if and only if it has an integer solution. Thus, one can equivalently define a linear potential function to be a function $\Phi(\mathbf{x}) = \sum_{S \in \Lambda} v_S \mathbf{x}(S)$ such that each $v_S \in \mathbb{N}$, i.e., we may assume $\Phi : \mathbb{N}^{\Lambda} \to \mathbb{N}$. In particular, since Φ is decreased by each reaction, it is decreased by at least 1.

The following theorem due to Czerner, Guttenberg, Helfrich, and Esparza, is crucial to proving limitations on execution bounded CRNs such as Theorem 6.5 and Theorem 6.7.

▶ **Theorem 6.3** ([6]). *A CRN has a linear potential function if and only if it is entirely execution bounded.*

6.2 Impossibility of stably deciding majority and parity

In this section, we prove Theorem 6.5, which is a special case of our main negative result, Theorem 6.7. We give a self-contained proof of Theorem 6.5 because it is simpler and serves as an intuitive warmup to some of the key ideas used in proving Theorem 6.7, without the complexities of dealing with arbitrary semilinear sets.

Theorem 6.5 shows a limitation on the computational power of entirely execution bounded, all-voting CRNs, but it requires an additional constraint on the CRN for the result to hold (and we later give counterexamples showing that this extra hypothesis is provably necessary), described in the following definition.

▶ **Definition 6.4.** *Let $\mathcal{D}$ be a CRD. The* output size *of $\mathcal{D}$ is the function $s : \mathbb{N} \to \mathbb{N}$ defined $s(n) = \min_{\mathbf{x},\mathbf{y}}\{\|\mathbf{y}\| \mid \mathbf{x} \Rightarrow \mathbf{y}, \|\mathbf{x}\| = n, \mathbf{x}$ is a valid initial configuration, $\mathbf{y}$ is stable$\}$, the size of the smallest stable configuration reachable from any valid initial configuration of size n. A CRD is* non-collapsing *if $\lim_{n\to\infty} s(n) = \infty$.*

Put another way, $\mathcal{D}$ is *collapsing* if there is a constant c such that, from infinitely many initial configurations $\mathbf{x}$, $\mathcal{D}$ can reach a stable configuration of size at most c. All population protocols are non-collapsing, since every reaction preserves the configuration size.

▶ **Theorem 6.5.** *No non-collapsing, all-voting, entirely execution bounded CRD can stably decide the majority predicate $[X_1 \geq X_2?]$ or the parity predicate $[X \equiv 1 \mod 2?]$.*

Proof. Let $\mathcal{D} = (\Lambda, R, \Sigma, \Upsilon_{\mathrm{Y}}, \Upsilon_{\mathrm{N}}, \mathbf{s})$ be a CRD obeying the stated conditions, and suppose for the sake of contradiction that $\mathcal{D}$ stably decides the majority predicate (so $\Sigma = \{X_1, X_2\}$).

We consider the sequence of stable configurations $\mathbf{a}_1, \mathbf{b}_1, \mathbf{a}_2, \mathbf{b}_2, \ldots$ defined as follows. Let $\mathbf{a}_1$ be a stable configuration reachable from initial configuration $\mathbf{s} + \{X_1, X_2\}$; since the correct answer is yes, all species present in $\mathbf{a}_1$ vote yes. Now add a single copy of X_2. By additivity, the configuration $\mathbf{a}_1 + \{X_2\}$ is reachable from $\mathbf{s} + \{X_1, 2X_2\}$, for which the correct answer in this case is no. Thus, since $\mathcal{D}$ stably decides majority, from $\mathbf{a}_1 + \{X_2\}$, a stable "no" configuration is reachable; call this $\mathbf{b}_1$. Now add a single X_1. Since the correct answer is yes, from $\mathbf{b}_1 + \{X_1\}$ a stable "yes" configuration is reachable, call it $\mathbf{a}_2$.

Continuing in this way, we have a sequence of stable configurations $\mathbf{a}_1, \mathbf{b}_1, \mathbf{a}_2, \mathbf{b}_2, \ldots$ where all species in $\mathbf{a}_i$ vote yes and all species in $\mathbf{b}_i$ vote no. Since $\mathcal{D}$ is non-collapsing, the size of the configurations $\mathbf{a}_i$ and $\mathbf{b}_i$ increases without bound as $i \to \infty$. (Possibly $\|\mathbf{a}_{i+1}\| < \|\mathbf{a}_i\|$, i.e., the size is not necessarily monotonically increasing, but for all sufficiently large $j > i$, we have $\|\mathbf{a}_j\| > \|\mathbf{a}_i\|$.) Since all species vote, for some constant $\delta > 0$, to get from $\mathbf{a}_i + \{X_2\}$ to $\mathbf{b}_i$, at least $\delta\|\mathbf{a}_i\|$ reactions must occur. This is because all species in $\mathbf{a}_i$ must be removed since they vote yes, and each reaction removes at most $O(1)$ molecules. (Concretely, let $\delta = 1/\max_{(\mathbf{r},\mathbf{p})\in R} \|\mathbf{r}\| - \|\mathbf{p}\|$, i.e., 1 over the most net molecules consumed in any reaction.) Similarly, to get from $\mathbf{b}_i + \{X_1\}$ to $\mathbf{a}_{i+1}$, at least $\delta\|\mathbf{b}_i\|$ reactions must occur.

Since $\mathcal{D}$ is entirely execution bounded, by Theorem 6.3, $\mathcal{D}$ has a linear potential function $\Phi(\mathbf{x}) = \mathbf{v} \cdot \mathbf{x}$, where $\mathbf{v} \geq \mathbf{0}$. Adding a single X_2 to $\mathbf{a}_i$ increases Φ by the constant $\mathbf{v}(X_2)$. Since $\|\mathbf{a}_i\|$ grows without bound, the number of reactions to get from $\mathbf{a}_i + \{X_2\}$ to $\mathbf{b}_i$ increases without bound as $i \to \infty$, and since each reaction strictly decreases Φ by at least 1, the total change in Φ that results from adding X_2 and then going from $\mathbf{a}_i + \{X_2\}$ to $\mathbf{b}_i$ is unbounded in i, so unboundedly negative for sufficiently large i (negative once i is large enough that $\delta\|\mathbf{a}_i\| \geq \mathbf{v}(X_2) + 2$). Similarly, adding a single X_1 to $\mathbf{b}_i$ and going from $\mathbf{b}_i + \{X_1\}$ to $\mathbf{a}_{i+1}$, the resulting total change in Φ is unbounded and (for large enough i) negative.

Φ starts this process at the constant $\Phi(\mathbf{s} + \{X_1, X_2\})$. Before $\|\mathbf{a}_i\|$ and $\|\mathbf{b}_i\|$ are large enough that $\delta\|\mathbf{a}_i\| \geq \mathbf{v}(X_2) + 2$ and $\delta\|\mathbf{b}_i\| \geq \mathbf{v}(X_1) + 2$ (i.e., large enough that the net change in Φ is negative resulting from adding a single input and going to the next stable configuration), Φ could increase, if $\Phi(\{X_1\})$ (resp. $\Phi(\{X_2\})$) is larger than the net decrease in Φ due to following reactions to get from $\mathbf{a}_i + \{X_2\}$ to $\mathbf{b}_i$ (resp. from $\mathbf{b}_i + \{X_1\}$ to $\mathbf{a}_i$).

However, since $\mathcal{D}$ is non-collapsing, this can only happen for a constant number of i (so Φ never reaches more than a constant above its initial value $\Phi(\mathbf{s} + \{X_1, X_2\})$), after which Φ strictly decreases after each round of this process. At some point in this process, $\mathcal{D}$ will not be able to reach all the way to the next $\mathbf{a}_i$ or $\mathbf{b}_i$ without Φ becoming negative, a contradiction.

The argument for parity is similar, but instead of alternating adding X_1 then X_2, in each round we always add one more X to flip the correct answer. ◀

Theorem 6.5 is false without the non-collapsing hypothesis. The following collapsing, leaderless (but all-voting and entirely execution bounded) CRD stably decides majority: Species X_1, x_1 vote yes, while X_2, x_2 vote no:

$$\begin{aligned} X_1 + X_2 &\to x_1 + x_2 \\ X_1 + x_2 &\to X_1 \\ X_2 + x_1 &\to X_2 \\ x_1 + x_2 &\to x_1 \end{aligned}$$

It has bounded executions from every configuration: $\min(\#X_1, \#X_2)$ of the first reaction can occur, and the other reactions decrease molecular count, so are limited by the total configuration size. However, it is collapsing since, for any n, there exists an input of size n that reaches a stable configuration of size 1. Theorem 6.5 is similarly false without the all-voting hypothesis; for each of the reactions with one product above, add another non-voting product W. This converts the CRD to be non-collapsing but not all-voting. Of course, the execution bounded hypothesis is also necessary: the original population protocols paper [1] showed that all-voting, non-collapsing, leaderless population protocols can stably decide all semilinear predicates.

The following collapsing, all-voting, leaderless (but entirely execution bounded) CRD stably decides parity. Let the input species be named X_1. Species X_1 votes yes, X_0 votes no:

$$\begin{aligned} X_1 + X_1 &\to X_0 \\ X_1 + X_0 &\to X_1 \\ X_0 + X_0 &\to X_0 \end{aligned}$$

6.3 Impossibility of stably deciding not eventually constant predicates

We now present our main negative result, Theorem 6.7, which generalizes Theorem 6.5 to show that such CRNs can stably decide only very limited (eventually constant) predicates.

▶ **Definition 6.6.** *Let $\phi : \mathbb{N}^d \to \{0, 1\}$ be a predicate. We say ϕ is* eventually constant *if there is $n_0 \in \mathbb{N}$ such that ϕ is constant on $\mathbb{N}^d_{\geq n_0} = \left\{\mathbf{x} \in \mathbb{N}^d \mid (\forall i \in \{1, \ldots, d\})\ \mathbf{x}(i) \geq n_0\right\}$, i.e., either $\phi^{-1}(0) \cap \mathbb{N}^d_{\geq n_0} = \emptyset$ or $\phi^{-1}(1) \cap \mathbb{N}^d_{\geq n_0} = \emptyset$.*

In other words, although ϕ may have an infinite number of each output, "sufficiently far from the boundary of the positive orthant" (where all coordinates exceed n_0), only one output appears. A complete proof appears in the full version of this paper.

▶ **Theorem 6.7.** *If a non-collapsing, all-voting, entirely execution bounded CRD stably decides a predicate ϕ, then ϕ is eventually constant.*

Proof sketch. This proof is similar to that of Theorem 6.5. In that proof, we repeatedly add a "constant amount of additional input $\{X_2\}$ or $\{X_1\}$, which flips the output". For more general semilinear, but not eventually constant, predicates, we dig into the structure of the semilinear set to find a sequence of constant-size vectors representing additional inputs that flip the correct output. Any predicate that is not eventually constant has infinitely many yes inputs and infinitely many no inputs, but in general they could be increasingly far apart: e.g., $\phi(\mathbf{x}) = 1$ if and only if $2^n \leq \|\mathbf{x}\| < 2^{n+1}$ for even n. For the potential function argument to work, each subsequent input needs to be at most a constant larger than the previous.

But if ϕ is *semilinear* (and not eventually constant) then we can show that there is a sequence of increasing inputs $\mathbf{x}_0 \leq \mathbf{x}_1 \leq \mathbf{x}_2 \leq \ldots$, each a *constant* distance from the next ($\|\mathbf{x}_{j+1} - \mathbf{x}_j\| = O(1)$), flipping the output ($\phi(\mathbf{x}_j) \neq \phi(\mathbf{x}_{j+1})$). Roughly, this is true for one of two reasons. Using Theorem 2.5, ϕ is a Boolean combination of threshold and mod sets. Either the mod sets are not combined to be trivially $\emptyset$ or $\mathbb{N}^d$, in which case we can find some vector $\mathbf{v}$ that, followed infinitely far from some starting point $\mathbf{x}_0$ (so $\mathbf{x}_i = \mathbf{x}_0 + i\mathbf{v}$) periodically hits both yes inputs ($\phi(\mathbf{x}_j) = 1$) and no inputs ($\phi(\mathbf{x}_j) = 0$). Otherwise, the mod sets can be removed and simplify the Boolean combination to only threshold sets, in which case the infinite sequence $\mathbf{x}_0, \mathbf{x}_1, \ldots$ can be obtained by moving along a threshold hyperplane that separates yes from no inputs. ◀

The statement of Theorem 6.5 does not mention the concept of a leader, but it would typically apply to leaderless CRDs. A CRD may be execution bounded from configurations with a single leader, but not execution bounded when multiple leaders are present (preventing the use of Theorem 6.3, which requires the CRD to be execution bounded from *all* configurations). For example, in Lemma 4.5, reaction (9) occurs finitely many times if the leader/voter S_Y or S_N has count 1. However, if S_Y and S_N can be present simultaneously (e.g., if we start with two leaders), then the reactions $S_Y + V_{NN} \to S_Y + V_{YN}$ and $S_N + V_{YN} \to S_N + V_{NN}$ can flip between V_{NN} and V_{YN} infinitely often in an unbounded execution.

If the CRN is leaderless, however, we have the following, which says that if it is execution bounded from *valid initial* configurations, then it is execution bounded from *all* configurations.

▶ **Lemma 6.8.** *If a leaderless CRD or CRC is execution bounded, then it is entirely execution bounded.*

Proof sketch. Since $\mathcal{C}$ is leaderless, the sum of two valid initial configurations is also valid. Thus if we can produce some species from a valid initial configuration, we can produce arbitrarily large counts of all species by adding up sufficiently many initial configurations. This means that for any configuration $\mathbf{x}$, from any sufficiently large valid initial configuration $\mathbf{i}$, some $\mathbf{y} \geqq \mathbf{x}$ is reachable from $\mathbf{i}$. But if $\mathcal{C}$ is execution bounded from $\mathbf{i}$, since $\mathbf{i} \Rightarrow \mathbf{y}$, it must also be execution bounded from $\mathbf{y}$, thus also from $\mathbf{x}$ since by additivity any reactions applicable to $\mathbf{x}$ are also applicable to $\mathbf{y}$. ◀

Lemma 6.8 lets us replace "entirely execution bounded" in Theorem 6.7 with "leaderless and execution bounded":

▶ **Corollary 6.9.** *If a non-collapsing, all-voting, leaderless, execution bounded CRD stably decides a predicate ϕ, then ϕ is eventually constant.*

In particular, since the original model of population protocols [1] defined them as leaderless and all-voting – and since population protocols are non-collapsing – we have the following.

▶ **Corollary 6.10.** *If an execution bounded population protocol stably decides a predicate ϕ, then ϕ is eventually constant.*

7 Conclusion

A key question remains open: *Can execution bounded CRNs compute semilinear functions and predicates within polylogarithmic time?* Angluin, Aspnes and Eisenstat [2] introduced a fast population protocol that simulates a register machine with high probability, and can be made probability 1 with semilinear predicates. However, this construction seems inherently unbounded in executions.

References

1 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *PODC 2004: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 290–299, 2004. doi:10.1145/1011767.1011810.

2 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, September 2008. doi:10.1007/S00446-008-0067-Z.

3 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007. doi:10.1007/S00446-007-0040-2.

4 Ho-Lin Chen, David Doty, Wyatt Reeves, and David Soloveichik. Rate-independent computation in continuous chemical reaction networks. *Journal of the ACM*, 70(3), May 2023. doi:10.1145/3590776.

5 Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. *Natural Computing*, 13(4):517–534, 2014. Preliminary version appeared in DNA 2012. doi:10.1007/s11047-013-9393-6.

6 Philipp Czerner, Roland Guttenberg, Martin Helfrich, and Javier Esparza. Fast and succinct population protocols for Presburger arithmetic. *Journal of Computer and System Sciences*, 140:103481, 2024. doi:10.1016/J.JCSS.2023.103481.

7 David Doty and Monir Hajiaghayi. Leaderless deterministic chemical reaction networks. *Natural Computing*, 14(2):213–223, 2015. Preliminary version appeared in DNA 2013. doi:10.1007/S11047-014-9435-8.

8 Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

9 S. Ginsburg and E. H. Spanier. Semigroups, Presburger formulas, and languages. *Pacific Journal of Mathematics*, 16(2):285–296, 1966.

10 Richard M Karp and Raymond E Miller. Parallel program schemata. *Journal of Computer and system Sciences*, 3(2):147–195, 1969. doi:10.1016/S0022-0000(69)80011-5.

11 Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, 1978. doi:10.1016/0304-3975(78)90036-1.

12 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7(4):615–633, 2008. doi:10.1007/s11047-008-9067-y.

Broadcast and Consensus in Stochastic Dynamic Networks with Byzantine Nodes and Adversarial Edges

Antoine El-Hayek
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

Monika Henzinger
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

Stefan Schmid
TU Berlin, Germany
Fraunhofer SIT, Berlin, Germany

Abstract

Broadcast and Consensus are most fundamental tasks in distributed computing. These tasks are particularly challenging in dynamic networks where communication across the network links may be unreliable, e.g., due to mobility or failures. Over the last years, researchers have derived several impossibility results and high time complexity lower bounds for these tasks. Specifically for the setting where in each round of communication the adversary is allowed to choose one rooted tree along which the information is disseminated, there is a lower as well as an upper bound that is linear in the number n of nodes for Broadcast and for $n \geq 3$ the adversary can guarantee that Consensus never happens. This setting is called the *oblivious message adversary for rooted trees*. Also note that if the adversary is allowed to choose a graph that does not contain a rooted tree, then it can guarantee that Broadcast and Consensus will never happen.

However, such deterministic adversarial models may be overly pessimistic, as many processes in real-world settings are stochastic in nature rather than worst-case.

This paper studies Broadcast on *stochastic* dynamic networks and shows that the situation is very different to the deterministic case. In particular, we show that if information dissemination occurs along random rooted trees and directed Erdős–Rényi graphs, Broadcast completes in $O(\log n)$ rounds of communication with high probability. The fundamental insight in our analysis is that key variables are mutually independent.

We then study two adversarial models, (a) one with Byzantine nodes and (b) one where an adversary controls the edges. (a) Our techniques without Byzantine nodes are general enough so that they can be extended to Byzantine nodes. (b) In the spirit of smoothed analysis, we introduce the notion of *randomized oblivious message adversary*, where in each round, an adversary picks $k \leq 2n/3$ edges to appear in the communication network, and then a graph (e.g. rooted tree or directed Erdős–Rényi graph) is chosen uniformly at random among the set of all such graphs that include these edges. We show that Broadcast completes in a finite number of rounds, which is, e.g., $O(k + \log n)$ rounds in rooted trees.

We then extend these results to All-to-All Broadcast, and Consensus, and give lower bounds that show that most of our upper bounds are tight.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Networks → Network algorithms

Keywords and phrases Broadcast, Smoothed Analysis, Stochastic Networks, Dynamic Networks

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.21

Related Version *Full Version*: `arXiv:2302.11988v2`

Funding *Antoine El-Hayek*: This project has received funding from the Austrian Science Fund (FWF) grant DOI 10.55776/P33775 with additional funding from the netidee SCIENCE Stiftung, 2020–2024.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 21; pp. 21:1–21:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Monika Henzinger: This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (MoDynStruct, No. 101019564) and the Austrian Science Fund (FWF) grant DOI 10.55776/Z422, grant DOI 10.55776/I5982, and grant DOI 10.55776/P33775 with additional funding from the netidee SCIENCE Stiftung, 2020–2024.
Stefan Schmid: This project has received funding from the German Research Foundation (DFG), SPP 2378 (project ReNO), 2023-2027.

1 Introduction

Broadcast and Consensus are two of most fundamental operations in distributed computing which, in large-scale systems, typically have to be performed over a *network*. These networks are likely to be dynamic and change over time due, e.g., to link failures, interference, or mobility. Understanding how information disseminates in such dynamic networks is hence important for developing and analyzing efficient distributed systems.

Over the last years, researchers have derived several important insights into information dissemination in dynamic networks. A natural and popular model assumes an *oblivious*[1] *message adversary* which controls the information flow between a set of n nodes, by dropping an arbitrary set of messages sent by some nodes in each round [7]. Specifically, the adversary is defined by a set of directed communication graphs, one per round, whose edges determine which node can successfully send a message to which other node in a given round. Based on this set of graphs, the oblivious message adversary chooses a sequence of graphs over time, one per round with repetitions allowed, in such a way that the time complexity of the information dissemination task at hand is maximized. This model is appealing because it is conceptually simple and still provides a highly dynamic network model: The set of allowed graphs can be arbitrary, and the nodes that can communicate with one another can vary greatly from one round to the next. It is, thus, well-suited for settings where significant transient message loss occurs, such as in wireless networks. As information dissemination is faster on dense networks, most literature studies oblivious message adversaries on sparse networks, in particular, on rooted trees [16, 30, 7, 21, 22]. In fact, it is easy to see that rooted trees are a minimal necessary requirement for a successful Broadcast and Consensus: if an adversary may choose a graph that does not contain a rooted tree, then it may forever prevent the dissemination of a piece of information.

Unfortunately, information dissemination can be slow in trees: Broadcast can take time linear in the number of nodes under the oblivious message adversary [16, 30], even for constant-height trees (as we show in the full version); and Consensus can even take super-polynomial time until termination, if it completes at all [7, 21]. Although this is bad news, one may argue that while the deterministic adversary model is useful in malicious environments, in real-word applications, the dynamics of communication networks is often more stochastic in nature. Accordingly, the worst-case model considered in existing literature may be overly conservative.

This motivates us, in this paper, to study information dissemination, and in particular Broadcast and Consensus tasks, in a scenario where the communication network is stochastic. Initially, we study a purely stochastic scenario where in each round, the communication network is chosen uniformly at random among all rooted trees. We then study several

[1] Note that the term oblivious here refers to the property that nodes are oblivious to who their neighbors are. However, our adversary is actually adaptive.

fundamental extensions of this model where the adversary has some limited control. In a first extension, we consider the case where some nodes (up to $\frac{2n}{3}$) may be Byzantine, that is, they may deviate arbitrarily from the protocol (and stop forwarding messages, for example). In a second extension, in the spirit of smoothed analysis, we study a setting where an adversary has some limited control over the communication network; we call this adversary the *randomized oblivious message adversary.* More specifically, we study the setting where first a worst-case adversary chooses k directed edges in the dynamic n-node network for some fixed k with $0 \leq k < \frac{2n}{3} - 1$[2], and then a rooted tree is chosen uniformly at random among the set of all rooted trees that include these edges.

We show that Broadcast completes within time $O(\log n)$ with high probability. We then show that this result even holds with Byzantine nodes. Under our randomized oblivious message adversary, Broadcast completes in $O(k + \log n)$ time with high probability.

It is useful to put our model into perspective with the SI (Susceptible-Infectious) model in epidemics [13]: while in the SI model interactions occur on a network that equals a clique, our model revolves around trees which are chosen by an adversary. This tree structure renders the analytical understanding of the information dissemination process harder, due to the lack of independence between the edges in the network in a particular round. A key insight from our paper is that we can prove the independence of a key variable, namely the increase in the number of "informed" nodes, which is crucial for our analysis. Our proof further relies on stochastic dominance, which makes it robust to the specific adversarial objective, and applies to any adversary definition (e.g., whether it aims to maximize the minimum or the expected number of rounds until the process completes).

We then extend our study to adversaries which are not limited to trees. In particular, we are interested in how the time complexity of Broadcast and Consensus depends on the density of the network. To this end, we consider *directed Erdős–Rényi graphs*, a directed version of the classic and well-studied random graphs. This graph family is parameterized by the number of edges m and hence allows us to shed light on the impact of the density. Specifically in this model, in each round the network is formed by sampling m edges. We again study two extensions: in the first extension some nodes behave as Byzantine nodes, while in the second extension, up to $k \leq m$ edges are chosen by an adversary, and then the remaining edges are sampled. While results for this model can be found in some cases where m is chosen so that the graph is an expander w.h.p. in each round by using the results from Augustine et al [2], in the case where m is small, our results are novel.

We show that all our results extend to multiple other problems, namely All-to-All Broadcast, Byzantine Consensus and Reliable Broadcast.

1.1 Model

Let n be the number of nodes, and let each node have a unique identifier from $[n]$. Time proceeds in a sequence of rounds $t = 1, 2, \ldots,$ such that in each round t the communication network is chosen according to one of the models defined below. In each round, every honest node sends a message to all of its out-neighbors before receiving one from its in-neighbor. There is no message size restriction. We will study the following models of communication:

[2] We can relax this condition to $k \leq (1 - \epsilon)n$ for a fixed parameter ϵ, which results in a multiplicative factor of $\frac{1}{\epsilon}$ in the running time.

Uniformly Random Trees

In the *Uniformly Random Trees* model, let $\mathcal{T}_n$ be the set of all directed rooted trees on n nodes (where all edges are pointed away from the root). In each round, the communication network is chosen uniformly at random among graphs in $\mathcal{T}_n$, independently from other rounds. All nodes are honest.

Uniformly Random Trees with Byzantine Nodes

In the *Uniformly Random Trees with Byzantine Nodes* model, in each round, the communication network is chosen uniformly at random among graphs in $\mathcal{T}_n$, independently from other rounds. We have $n - f$ honest nodes, and f nodes are Byzantine, that is, they might behave arbitrarily (and even coordinate to make the protocol fail). We assume access to cryptographic tools that allow nodes to sign and encrypt messages. We restrict $f \leq \frac{2n}{3} - 1$.

Uniformly Random Trees with Adversarial Edges

In the *Uniformly Random Trees with Adversarial Edges* model, in each round, the communication network is chosen as follows: A randomized oblivious message adversary chooses k directed edges, then a graph is chosen uniformly at random among all graphs in $\mathcal{T}_n$ that include those k edges, and the choise is independent from other rounds. All nodes are honest. We restrict $k \leq \frac{2n}{3} - 1$.

Directed Erdős–Rényi graphs

In the *directed Erdős–Rényi graphs* model, let $m \in [n^2]$. In each round, the communication network is chosen by uniformly sampling without replacement m edges out of the possible n^2 edges of the graph, independently from other rounds. All nodes are honest.

Directed Erdős–Rényi graphs with Byzantine Nodes

In the *directed Erdős–Rényi graphs with Byzantine nodes* model, let $m \in [n^2]$. In each round, the communication network is chosen by uniformly sampling without replacement m edges out of the possible n^2 edges of the graph, independently from other rounds. We have $n - k$ honest nodes, and k nodes are Byzantine, that is, they might behave arbitrarily (and even coordinate to make the protocol fail). We assume access to cryptographic tools that allow nodes to sign and encrypt messages. We restrict $k < \frac{2n}{3}$.

Directed Erdős–Rényi graphs with Adversarial Edges

In the *directed Erdős–Rényi graphs with Adversarial Edges* model, let $0 \leq k \leq m \leq n^2$. In each round, the communication network is chosen as follows: A randomized oblivious message adversary chooses k edges, $m - k$ edges are sampled without replacement out of the remaining $n^2 - k$ edges. All nodes are honest. We restrict $k < \frac{3}{4}n^2$.

In those models, we will study the following problems:

Broadcast

For the *Broadcast*[3] problem, we start by giving a message to *one* (honest) node. Each honest node that received the message will replicate it as many times as needed, and start forwarding it to its neighbors[4]. Then Broadcast *completes* when the message has been forwarded to all other nodes.

All-to-All Broadcast

In the *All-to-All Broadcast* problem, we start by giving a distinct message to *each* node. Each honest node that received a message will replicate it as many times as needed, and start forwarding it as well. Then All-to-All Broadcast *completes* when each honest node receives a copy of every message. In each round, each honest node forwards all the messages it has received in previous rounds to all its out-neighbors.

Consensus

In the *Consensus* problem, we start by giving a value $v_p \in \{0, 1\}$ to each node p, and Consensus completes when each honest node decided on a value in $\{0, 1\}$. This should satisfy the following conditions:

- **Agreement:** No two honest nodes decide differently.
- **Termination:** Every honest node eventually decides.
- **Validity:** The value the honest nodes agree on should be one of the input values v_p.

1.2 Our Results

We study Broadcast in the above mentioned models, then apply those results to All-to-All broadcast and Consensus. We prove the following theorems:

▶ **Theorem 1.** *For any $c \geq 1$ and $n \geq 5$, Broadcast on Uniformly Random Trees completes within $32 \cdot c \cdot \ln n$ rounds with probability $p > 1 - \frac{1}{n^c}$.*

We also show that these results are asymptotically tight. Indeed, we cannot hope for a similar probability for a number of rounds that is $o(\ln n)$:

▶ **Theorem 2.** *If $n \geq 2$, then the probability that Broadcast (and All-to-All Broadcast) on Uniformly Random Trees fails to complete within $\log n$ rounds is at least $\frac{1}{4}$.*

We have similar results for all the combinations of model and problem, which we summarize in Table 1.

Applications

Our results have some interesting applications. In an idea similar to Ghaffari, Kuhn and Su's work [23], All-to-All Broadcast allows us, e.g., to implement algorithms that run on a clique in a synchronous setting in our sparser graphs. Indeed, if All-to-All Broadcast needs

[3] The Broadcast problem can also be seen as computing the *dynamic eccentricity* of the source node. Other flavors of Broadcast have also been studied under the name *dynamic radius* [20].

[4] This is known as “flooding” or “rumor passing”

	Broadcast	All-to-All Broadcast	Consensus
Uniformly Random Trees (URT)	$O(c \cdot \log n), q \leq n^{-c}$ $\Omega(\log n)$	$O(c \cdot \log n), q \leq n^{1-c}$ $\Omega(\log n)$	$O(c \cdot \log n), q \leq n^{-c}$
URT with Byzantine Nodes	$O(c \cdot \log n), q \leq n^{-c}$ $\Omega(\log n)$	$O(c \cdot \log n), q \leq n^{1-c}$ $\Omega(\log n)$	$O(f \cdot c \cdot \log n), q \leq n^{-c}$
URT with Adversarial Edges	$O(c \cdot (\log n + k)), q \leq n^{-c}$ $\Omega(\log n + k)$	$O(c \cdot (\log n + k)), q \leq n^{1-c}$ $\Omega(\log n + k)$	$O(c \cdot (\log n + k)), q \leq n^{-c}$
Directed Erdős–Rényi graphs (DER)	$O\left(\left\lceil \frac{c}{m/n} \right\rceil \log n\right)$,$q \leq n^{-c} \log n$ $O\left(\frac{c \log n}{\log(1+\frac{m}{n})}\right)$ if $\frac{m}{n} \geq \ln n$ with $q \leq n^{-c} \log n$ $\Omega\left(\frac{\log n}{\log(1+m/n)}\right)$	$O\left(\left\lceil \frac{c}{m/n} \right\rceil \log n\right)$, $q \leq n^{1-c} \log n$ $O\left(\frac{c \log n}{\log(1+\frac{m}{n})}\right)$ if $\frac{m}{n} \geq \ln n$ with $q \leq n^{1-c} \log n$ $\Omega\left(\frac{\log n}{\log(1+m/n)}\right)$	$O\left(\left\lceil \frac{c}{m/n} \right\rceil \log n\right)$, $q \leq n^{-c} \log n$ $O\left(\frac{c \log n}{\log(1+\frac{m}{n})}\right)$ if $\frac{m}{n} \geq \ln n$ with $q \leq n^{-c} \log n$
DER with Byzantine Nodes	$O\left(\left\lceil \frac{c}{m/n} \right\rceil \log n\right)$, $q \leq n^{-c} \log n$ $\Omega\left(\frac{\log n}{\log(1+m/n)}\right)$	$O\left(\left\lceil \frac{c}{m/n} \right\rceil \log n\right)$, $q \leq n^{1-c} \log n$ $\Omega\left(\frac{\log n}{\log(1+m/n)}\right)$	$O\left(f \cdot \left\lceil \frac{c}{m/n} \right\rceil \log n\right)$, $q \leq n^{-c} \log n$
DER with Adversarial Edges	$O\left(\left\lceil \frac{c \cdot (n^2-k)}{(m-k)n} \right\rceil \log n\right)$ with $q \leq n^{-c} \log n$ $\Omega\left(\frac{\log n}{\log(1+m/n)}\right)$	$O\left(\left\lceil \frac{c \cdot (n^2-k)}{(m-k)n} \right\rceil \log n\right)$ with $q \leq n^{1-c} \log n$ $\Omega\left(\frac{\log n}{\log(1+m/n)}\right)$	$O\left(\left\lceil \frac{c \cdot (n^2-k)}{(m-k)n} \right\rceil \log n\right)$ with $q \leq n^{-c} \log n$

Figure 1 Our main results, where $c > 0$ is any constant and q is the failure probability.

R rounds to complete with high probability, then each round of communication of a clique can be simulated by R rounds of Uniformly Random Trees with high probability. Essentially, if an algorithm runs in T rounds, with $T \leq n^{c-1}$, in a clique network, we can implement it with high probability in $R \cdot T$ rounds in the Uniformly Random Trees network, which is essentially a logarithmic overhead. In particular, in the Uniformly Random Trees with Byzantine Nodes model, we have:

▶ **Theorem 3.** *Let $\mathcal{A}$ be a distributed synchronous algorithm that runs on a static clique in T rounds, where $T \leq \alpha n^x$ for some constant $\alpha, x \in \mathbb{R}_+$, and has a probability of success p. Assume $\mathcal{A}$ is robust to f Byzantine nodes, and $f \leq \frac{2}{3}n - 1$. Then, assuming standard cryptographic tools*[5]*, there exists a distributed algorithm $\mathcal{A}'$ that runs on Uniformly Random Trees in $T \cdot 144 \cdot \log n \cdot c$ rounds, and has a probability of success $p' \geq p(1 - \alpha n^{1+x-c})$, for any $c \geq 1 + x$. Moreover, $\mathcal{A}'$ is robust to f Byzantine nodes.*

In particular, we can apply known results on reliable Broadcast and Byzantine Consensus to show the following results:

▶ **Corollary 4.** *For any $c \geq 1$, and $f \leq \frac{2}{3}n - 1$, in the Uniformly Random Trees with f Byzantine nodes, there exists an algorithm for Reliable Broadcast, that is robust to f Byzantine nodes, that runs in $(f + 1) \cdot 144 \cdot c \cdot \log n$ rounds, and succeeds with probability $p \geq 1 - n^{2-c}$.*

▶ **Corollary 5.** *For any $c \geq 1$ and $f < \frac{n}{3}$, in the Uniformly Random Trees with f Byzantine nodes, there exists an algorithm for Byzantine Consensus, that is robust to f Byzantine nodes, that runs in $3(f + 1) \cdot 144 \cdot c \cdot \log n$ rounds, and succeeds with probability $p \geq 1 - 2n^{2-c}$.*

Throughout the paper, the filtration of the process is denoted as $\{\mathcal{F}_t\}_{t \in \mathbb{N}}$, that is, $\mathcal{F}_t$ is the amount of information available after timestep t.

[5] Specifically, our approach requires authenticated messages. Encryption may also be needed, only if the protocol $\mathcal{A}$ is vulnerable to eavesdropping. Both can be implemented using standard cryptographic tools.

Organization

The paper is organized as follows. First, we review related work in Section 2. Then, due to space restrictions, we only give a technical overview in Section 3, as further details can be found in the full version of the paper. In this overview we first discuss a new result on the number of rooted trees containing a certain set of edges, then discuss how we analyzed information dissemination in random trees first, and finally in directed Erdős–Rényi graphs.

2 Related Work

Information dissemination in general and Broadcasting and Consensus in particular are fundamental topics in distributed computing. In contrast to this paper, most classic literature on network Broadcast as well as on related tasks such as gossiping and Consensus, considers a static setting, e.g., where in each round each node can send information to one neighbor [24, 19].

Especially the Byzantine setting has received much attention in the literature. Important results include Dolev and Strong [12] on reliable Broadcast which is robust to f Byzantine nodes, and runs in $T = f + 1$ rounds, or Berman, Garay and Perry [3] on King's algorithm that solves reliable Broadcast, is robust to f Byzantine nodes, and runs in $T = 3(f + 1)$ rounds. To just name a few.

In terms of dynamic networks, Kuhn, Lynch and Oshman [25] explore the all-to-all data dissemination problem (gossiping) in an undirected setting, where nodes do not know beforehand the total number of nodes and must decide on that number. Dutta, Pandurangan, Rajaraman, Sun and Viola [14] generalize the model to when not all nodes need to forward their message, but only k tokens must be forwarded. Augustine, Pandurangan, Robinson and Upfal [2] show that if the graph is an expander in every round, broadcast is complete within $O(\log n)$ rounds, even if a small enough constant fraction of nodes get churned in each round. Ahmadi, Kuhn, Kutten, Molla and Pandurangan [1] study the message complexity of Broadcast also in an undirected dynamic setting, where the adversary pays up a cost for changing the network.

In dynamic networks, the oblivious message adversary is a commonly considered model, especially for Broadcast and Consensus problems, first introduced by Charron-Bost and Schiper [5]. The Broadcast problem under oblivious message adversaries has been studied for many years. A first key result for this problem was the $n \log n$ upper bound by Zeiner, Schwarz, and Schmid [30] who also gave a $\lceil \frac{3n-1}{2} \rceil - 2$ lower bound. Another important result is by Függer, Nowak, and Winkler [20] who presented an $O(\log \log n)$ upper bound if the adversary can only choose nonsplit graphs; combined with the result of Charron-Bost, Függer, and Nowak [4] that states that one can simulate $n - 1$ rounds of rooted trees with a round of a nonsplit graph, this gives the previous $O(n \log \log n)$ upper bound for Broadcasting on trees. Dobrev and Vrto [10, 9] give specific results when the adversary is restricted to hypercubic and tori graphs with some missing edges. El-Hayek, Henzinger, and Schmid [15, 16] recently settled the question about the asymptotic time complexity of Broadcast by giving a tight $O(n)$ upper bound, also showing the upper bound still holds in more general models. Regarding Consensus, Coulouma, Godard and Peters in [7] presented a general characterization on which dynamic graphs Consensus is solvable, based on Broadcastability. Winkler, Rincon Galeana, Paz, Schmid, and Schmid [21] recently presented an explicit decision procedure to determine if Consensus is possible under a given adversary, enabling a time complexity analysis of Consensus under oblivious message adversaries, both for a centralized decision procedure as well as for solving distributed Consensus. They also showed that reaching Consensus under an oblivious message adversary can take exponentially longer than Broadcasting.

In contrast to the above works, in this paper we study a more randomized message adversary, considering a stochastic model where adversarial graphs are partially chosen uniformly at random. While a randomized perspective on dynamic networks is natural and has been considered in many different settings already, existing works on random dynamic communication networks, e.g., on the radio network model [17], on rumor spreading [6], as well as on epidemics [13], do not consider oblivious message adversaries. Note, however, that the information dissemination considered in this paper is similar to the SI model for virus propagation, with results having implications in both directions [18]. For example, Doerr and Fouz [11] introduced an information dissemination protocol inspired by epidemics. More generally, randomized information dissemination protocols can be well-understood from an epidemiological point-of-view, and are very similar to the SI model which has been very extensively studied. In contrast to the typical SI models considered in the literature [28], however, our model in this paper revolves around tree communication structures which introduce additional technical challenges. Furthermore, existing literature often provides results in expectation, while we in this paper provide tail bounds.

Many papers have tried to bridge the gap between the deterministic and random case, using smoothed analysis. In [27], Meir, Paz and Schwartzman study the broadcast problem in noisy networks, under different definitions on noise. In particular, if in each round the graph given by the adversary is replaced by a graph chosen uniformly at random among graphs at hamming distance at most k from the original graph, in the case where the adversary can suggest any connected graph, then Broadcast is reduced from n rounds to $O(\min\{n, n\sqrt{\frac{\log n}{k}}\})$ rounds, in the case of an adaptive adversary. If the adversary is oblivious, then Dinitz, Fineman, Gilbert and Newport [8] showed that it is further reduced to $O(n^{2/3}/k^{1/3} \times \log n)$.

3 Technical Overview

Our paper contains a conceptional contribution, namely the extension of the notion of oblivious message adversary in a natural way to a randomized setting that limits the power of the adversary, as well as two technical contributions. We explained already the conceptional contribution in the introduction, and we sketch in this section now the main technical contributions of our paper. They are of graph theoretical as well as algorithmic nature. (1) On the graph theoretic side, we show a new result on the number of rooted trees that satisfy a certain property. (2) On the algorithmic side we show how to use this result to give an upper bound on the number of rounds for the models introduced in the introduction. Note that we study both the conventional as well as the Byzantine setting, where faulty nodes can stop forwarding, send wrong messages, and even coordinate to make the protocol fail. However, we assume access to cryptographic tools so that is used by each node to sign its messages. Thus, when receiving a message, nodes can be confident about the sender of each message and its content.

3.1 Counting rooted trees

Given a graph consisting of n vertices together with a directed rooted forest F of e edges on them, Pitman [29] showed in 1999 that there are n^{n-1-e} many directed rooted trees over these vertices that contain F. While useful, this result is not sufficient for our purposes as we need to count the number of trees with a given node v as root.

Thus, we show the following extended result:

▶ **Theorem 6.** *Let us be given a directed rooted forest F on n vertices, let $v \in [n]$ be the root of a component in F, and f be the number of vertices of that component (note that we can have $f = 1$ if v is an isolated vertex). Then the number of directed rooted trees T on n vertices, such that F is contained in T, and such that v is the root of T, is $fn^{n-2-|E|}$.*

Note that our result implies the prior result.

To show our result, we develop techniques which differ significantly from Pitman's proof. Indeed, Pitman relies on the symmetry of the vertices in the rooted tree. However, for our result, the symmetry is broken as one vertex is different from the other with the new requirement that it is the root. We hence make use of another type of symmetry in the trees in our analysis that is based on group actions.

We first ignore the orientations of the edges in F and find the set A_F of all undirected trees that contain F. We can compute the cardinality of that set with a result by Lu, Mohr and Székely [26]. We then root each of those trees at v. This will give a direction to every edge that might or might not agree with its direction in F. We now want to partition A_F into subsets such that all subsets have the same size and only one tree from each subset has edges that agree with the direction of F. The number we are looking for is then the number of subsets, which is the ratio between the cardinality of A_F and the size of the subsets.

To create the subsets, we introduce a specific group tailored to F, and an action of that group on A_F. It is known that the set of all orbits of the action partition A_F, and we show that exactly one element in each orbit has edges in the same direction as F. To see unicity, we take an element T of A_F that has edges in the same direction as F, and take an element $T' \neq T$ in its orbit, that is there exists a nontrivial group element g such that T' is obtained from T by applying the action of g to T. We show that this action must change the direction of at least one edge of F, and thus T' does not have edges in the same direction as F. For existence, we show that for every $T \in A_F$, we can find a group element g such that, if applied to T, yields a tree that has edges in the same direction as F. We then show how to compute the size of each orbit. This allows us to deduce the number of orbits, which equals the number of trees that we want to count.

3.2 Analysis of the information dissemination

The main technical challenge is to analyze Broadcast in uniformly random trees (URTs) and in directed Erdős–Rényi graphs (DERs). Our techniques for both types of graphs are general and can be extended to adversarial settings, i.e. Byzantine nodes or adversarial edges, as well as to all-to-all Broadcast and Consensus. We only discuss Broadcast in this overview and give the technical details for all models in the subsequent sections.

Random Trees

Our analysis for URTs proceeds in steps. (A) First we analyze the uniformly random tree model, i.e., the model where the adversary controls none of the edges. (B) Second we allow adversarial, i.e., Byzantine, nodes in the uniformly random tree model. (C) Third we analyze the randomized oblivious message adversary with parameter k.

We next sketch the main challenges and how to overcome them. We use n to denote the number of nodes, I_t, resp. S_t to denote the set of informed, resp. uninformed nodes after round t, and set $N_t = |I_t|$.

(A) When choosing a rooted tree T uniformly at random, there is a high dependence between the events that indicate whether an edge belongs to T or not. Assume that nodes 1, 2, and 3 as well as the edges (1,2) and (2,3) belong to T. Then the edge (1,3) cannot belong to T. Still, we are able to show that for every node $i \in S_t$ the probability that it is informed in round t is N_t/n, independently of whether other nodes are informed or not in round t, using the tree counting results discussed before, i.e., $\Delta_t := N_{t+1} - N_t$ follows a binomial distribution with parameters $(n - N_t, N_t/n)$:

▶ **Lemma 7.** *For any $t > 0$, conditioned on N_t $N_{t+1} - N_t$ follows a binomial distribution with parameters $\left(n - N_t, \frac{N_t}{n}\right)$.*

Thus, in expectation, Δ_t is $(n - N_t)N_t/n$. Now assume for the moment that each round would perform according to its expectation. Then as long as $N_t \leq n/2$, $(n - N_t)N_t/n \geq N_t/2$, i.e., the number of informed nodes increases by a multiplicative factor of at least $3/2$ in each round and, thus, there are $O(\log n)$ many rounds. As soon as $N_t > n/2$ then $(n - N_t)N_t/n \geq (n - N_t)/2$, i.e., the distance between the maximum number n and the current number N_t of informed nodes is halved, and, thus, there are at most $O(\log n)$ many rounds.

However, N_t will not increase in every round according to its expectation. Thus, to make this intuition formal we define a random variable X_t for each round t with $X_0 = 1$ that increases by $(n - X_t)X_t/n$ if Δ_t is at least by its expected value (such a round is called an *increasing* round) and X_t remains unchanged otherwise. It follows from the definition of X_t that it increases monotonically, never reaches n, and always lower bounds N_t. The number of increasing rounds needed for X_t to reach a value larger than $n - 1$ is at most $2 \ln n$, by a similar argument to the one above. It remains to show that X_t increases frequently. We show that the probability that X_t increases in a round is larger than $1/4$, as the binomial variable Δ_t has a probability larger than $1/4$ to be at least at its expectation. Then, Hoeffding's inequality for binomial distributions shows that with probability at least $1 - n^{-c}$ there are more than $2 \ln n$ increasing rounds within the first $32c \ln n$ rounds giving the desired upper bound:

▶ **Theorem 1.** *For any $c \geq 1$ and $n \geq 5$, Broadcast on Uniformly Random Trees completes within $32 \cdot c \cdot \ln n$ rounds with probability $p > 1 - \frac{1}{n^c}$.*

We also show that the bound is asymptotically tight by proving that with constant probability at least $\log n$ rounds are needed. To do so let $Z_t := X_{u_t}$, where u_t is the number of increasing rounds up to round t. Thus, intuitively Z_t is X_t with non-increasing rounds omitted. We first show inductively that $\mathbb{E}[N_t] \leq Z_t$. The intuitive reason is that initially $Z_0 = N_0 = \mathbb{E}[N_t]$ and, inductively, in each round Z_t increases by at least as much as $\mathbb{E}[N_t]$. Then we show by induction that $Z_t = n(1 - (n - 1/n))^{2^t}$, which implies that $Z_{\log n} = n(1 - (n - 1/n))^n \leq n(1 - 1/4) = 3n/4$. Thus, $\mathbb{E}[N_{\log n}] \leq 3n/4$ and the lower bound follows by applying Markov's inequality:

▶ **Theorem 2.** *If $n \geq 2$, then the probability that Broadcast (and All-to-All Broadcast) on Uniformly Random Trees fails to complete within $\log n$ rounds is at least $\frac{1}{4}$.*

(B) We extend the above model by allowing $f < 2n/3$ *Byzantine nodes* that might forward wrong or no messages, and that can coordinate to make the protocol fail. The process that chooses the communication network, i.e., the random tree, does not know which nodes are Byzantine and, thus, they are part of the network as before, i.e., the tree still consists of n nodes. Furthermore, we assume access to cryptographic tools so that every node can be

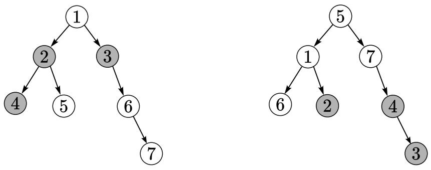


Figure 2 Shaded nodes are informed nodes. The adversary will choose the right tree over the left tree.

confident about the sender of each message and its content. Here the goal is to inform all $n - f$ honest nodes, i.e., it does not matter whether the Byzantine nodes are informed or not. Almost the same argument as for (A) shows that $N_{t+1} - N_t$ follows a binomial distribution with parameters $(n - f - N_t, N_t/n)$ and also the rest of the analysis, including the lower bound go through.

▶ **Theorem 8.** *For any $c \geq 1$, and $f \leq \frac{2}{3}n - 1$, Broadcast on Uniformly Random Trees with f Byzantine nodes completes within $144 \cdot c \cdot \log n$ rounds with probability $p > 1 - \frac{1}{n^c}$.*

(C) In the *uniformly random trees with adversarial edges model* an adversary chooses first up to k directed edges and then a random tree containing these edges is selected. As before we want to show that the probability that an uninformed node i is informed in round t is independent from other uninformed nodes being informed. This, however, is only true if the adversary uses a *specific optimal* strategy. For an example where the probabilities are not independent, consider a graph with 4 nodes, 2 informed and 2 uninformed. If the adversary introduces an edge from each uninformed node to a different informed node, then for each uninformed node the probability that it is informed in the tree of this round is 1/4. However, the probability that both uninformed nodes are informed in the tree of this round is zero, as only one random edge can be added, which will cause at most one uninformed node to become informed.

(C1) Thus, we first determine the optimal strategy for the adversary: Recall that the adversary wants to maximize the number of rounds. As we show, this implies that a greedy strategy, where the adversary minimizes the increase of N_t in each round t, is an optimal strategy for the adversary. To do so, we use a coupling argument comparing the number of informed nodes of the greedy strategy to a non-greedy strategy and showing that a greedy strategy informs all n nodes no later than a non-greedy strategy.

▶ **Lemma 9** (Distribution Domination). *Let t be a round. Let E_1, E_2 be two sets of edges the adversaries could choose for round t. Let $N_t^{(1)}$ (resp. $I_t^{(1)}$) be the number (resp. set) of informed nodes after round t if E_1 is chosen, and $N_t^{(2)}$ (resp. $I_t^{(2)}$) if E_2 is chosen. Then if $\mathbb{P}(N_t^{(1)} \geq m) \geq \mathbb{P}(N_t^{(2)} \geq m)$ for every $m \in \mathbb{N}$ (that is, if $N_t^{(1)}$ stochastically dominates $N_t^{(2)}$), then choosing E_2 is a better strategy for the adversary than choosing E_1.*

Next we analyze what edges are selected by a greedy strategy using three steps: (a) As an edge from an informed node to an uninformed node causes the uninformed node to be informed, the greedy strategy will never put such an edge. Thus, the adversary will only construct trees that do not contain such edges, which we call *non-increasing* trees. This is illustrated in Figure 2.

(b) We show that there is no advantage for the adversary to choose multiple trees. To show this we use a carefully chosen merge operation between any two non-increasing trees that guarantees that the resulting tree is non-increasing together with our new counting theorem for rooted trees. Thus, we can assume that the greedy strategy that is chosen always chooses just one non-increasing rooted tree, which we call U. This is illustrated in Figure 3.

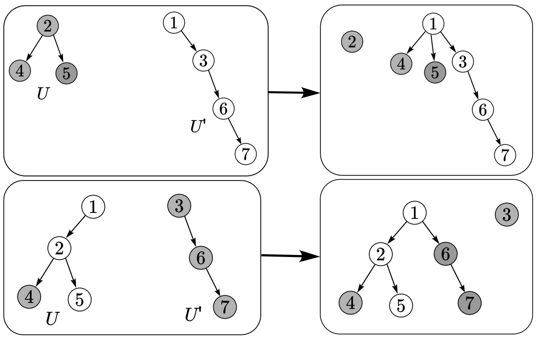


Figure 3 Merging examples. The adversary will always choose the right option over the left one.

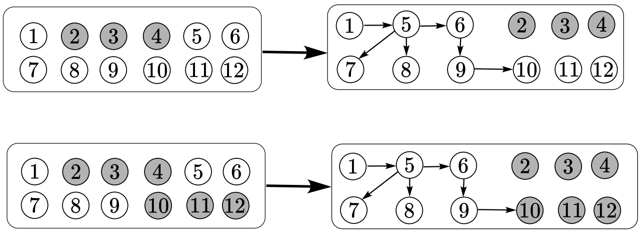


Figure 4 The best strategy for the adversary A, with $k = 6$. Shaded nodes are informed nodes. In the top example, nodes $5, 6, 7, 8, 9$ and 10 are safe from being informed, whereas node 1 can still be informed. In the bottom example, nodes $5, 6, 7, 8$, and 9 are safe, whereas node 1 can still be informed. However, node 1 is safe from being informed by node 10.

(c) We then argue that U should contain as many uniformed nodes as possible. The basic intuition is that if an uninformed node is the child of another uninformed node, it cannot become informed in this round, i.e., it is "protected". Given k edges, the adversary "protects" as many uninformed nodes as possible by building U from $\min(|S_t|, k+1)$ uninformed and $\max(k+1-|S_t|, 0)$ informed nodes. The fact that U is also non-increasing implies that the root of U is an uninformed node. This gives the optimal strategy, illustrated in Figure 4. We show all the above steps using stochastic dominance.

(C2) Finally we show that with this specific optimal strategy, the adversary can only force $O(k + \log n)$ many rounds with high probability. It follows that with high probability, the adversary cannot achieve more rounds with any other - optimal or non-optimal - strategy. To do so, we break the rounds into phases: (i) The first phase consists of all rounds where $|S_k| \geq k+1$. In this case, the adversary forms one tree with $k+1$ uninformed nodes and there are $|S_k| - k - 1 = n - N_t - k - 1$ additional uniformed singleton nodes, as well as N_t informed singleton nodes in the forest. Thus, we can apply exactly the same argument as in (A) to show that $N_{t+1} - N_t$ follows a binomial distribution with parameters $(n - k - N_t, N_t/n)$. (ii) The second phase consists of all rounds where $|S_k| \leq k$. Thus U consists of all uninformed nodes and at least one informed node. Thus, $N_{t+1} - N_t$ can increase by at most 1, namely if the root s of U receives a parent in the tree, and, using our new counting theorem for rooted trees, we show that the probability of that is $(N_t - (k + 1 - |S_t|))/n = (n - k - 1)/n$, i.e. $N_{t+1} - N_t$ is a binomial distribution with parameters $(1, (n - k - 1)/n)$. Using Hoeffding's inequality for binomial distributions similar to (A) we then show the result:

▶ **Theorem 10.** *If the adversary controls k edges in each round, for $k \leq \frac{2}{3}n - 1$, then for any $c \geq 1$, with probability $p \geq 1 - n^{-c}$, Broadcast completes within $O(k + \log n)$ rounds.*

Directed Erdős–Rényi graphs

Directed Erdős–Rényi graphs consist of m edges chosen uniformly at random among the n^2 potential edges. Intuitively they have less structure than uniformly random trees, which makes the analysis of Broadcast simpler. We present the main ideas below. Note that we also analyze Byzantine nodes and adversarial edges in that model, but omit these extensions in this overview.

Sampling a directed Erdős–Rényi graph is equivalent to choosing m edges *without* replacement from the set of all possible edges. We call that Scheme 1. Then we observe, using a coupling argument, that Scheme 1 requires no more rounds than Scheme 2, where in each round m edges are chosen *with* replacement. Finally, to analyze Scheme 2, we basically partition the sequence of rounds of Scheme 2 into $2\lceil(\log n)/2\rceil$ phases, such that for each of the first $\lceil(\log n)/2\rceil$ phases the number of informed nodes doubles in each phase and for each of the last $\lceil(\log n)/2\rceil$ phases the number of uninformed nodes halves in each phase. Note that Broadcast completes after the last phase. Using Hoeffding's inequality for binomial distributions we show that phase i for $1 \leq i \leq \lceil\log n/2\rceil$ requires with high probability at most $O(\max\{\log n, 2^{i-1}\}n/2^{i-1})$ sampled edges, and, thus, $O(\lceil\max\{\log n, 2^{i-1}\}/(2^{i-1}m/n)\rceil)$ rounds, and for $\lceil\log n/2\rceil + 1 \leq i \leq 2\lceil\log n/2\rceil$ phase i requires with high probability at most $O(\max\{\log n, 2^{j-2}\}n/2^{j-1})$ sampled edges with with $j := 2\lceil\log n/2\rceil - i$, and, thus, $O(\lceil\max\{\log n, 2^{j-1}\}/(2^{j-1}m/n)\rceil)$ rounds. Summed over all phases this shows that with high probability $O(\lceil n/m\rceil\log n)$ rounds suffice for Scheme 2 to reach Broadcast. Note that the analysis extends to the setting when the graph in each round contains *at least* m edges. We also show that a lower bound that implies that this upper bound is tight for $m \leq n$. We also give somewhat different analysis where the number of informed resp. uninformed nodes does not double, but increases by $(1 + m/n)$ that is tight for $m \geq n \ln n$. Our results can thus be summarized by the following theorems:

▶ **Theorem 11.** *For any $c \geq 1$, in scheme 2, and therefore scheme 1, Broadcast completes within $O\left(\left\lceil\frac{cn}{m}\right\rceil\log n\right)$ rounds with probability $p \geq 1 - n^{-c}\log n$.*

▶ **Theorem 12.** *For any $c \geq 1$ and $m \in [n^2]$ such that $m/n \geq \ln n$, in scheme 2 and in scheme 1, Broadcast completes within $O\left(\frac{c\cdot\log n}{\log(1+m/n)}\right)$ rounds with probability $p \geq 1 - n^{-c}\log n$.*

▶ **Theorem 13.** *In scheme 1, and thus in scheme 2, Broadcast fails to complete within $\frac{\log(n)-1}{\log(1+m/n)}$ rounds with probability at least $\frac{1}{2}$.*

References

1 Mohamad Ahmadi, Fabian Kuhn, Shay Kutten, Anisur Rahaman Molla, and Gopal Pandurangan. The communication cost of information spreading in dynamic networks. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*, pages 368–378. IEEE, 2019. doi:10.1109/ICDCS.2019.00044.

2 John Augustine, Gopal Pandurangan, Peter Robinson, and Eli Upfal. Towards robust and efficient computation in dynamic peer-to-peer networks. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 551–569. SIAM, 2012. doi:10.1137/1.9781611973099.47.

3 Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 410–415. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63511.

4 Bernadette Charron-Bost, Matthias Függer, and Thomas Nowak. Approximate consensus in highly dynamic networks: The role of averaging algorithms. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 528–539. Springer, 2015. doi:10.1007/978-3-662-47666-6_42.

5 Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Comput.*, 22(1):49–71, 2009. doi:10.1007/s00446-009-0084-6.

6 Andrea E. F. Clementi, Pierluigi Crescenzi, Carola Doerr, Pierre Fraigniaud, Francesco Pasquale, and Riccardo Silvestri. Rumor spreading in random evolving graphs. *Random Struct. Algorithms*, 48(2):290–312, 2016. doi:10.1002/rsa.20586.

7 Étienne Coulouma, Emmanuel Godard, and Joseph G. Peters. A characterization of oblivious message adversaries for which consensus is solvable. *Theor. Comput. Sci.*, 584:80–90, 2015. doi:10.1016/j.tcs.2015.01.024.

8 Michael Dinitz, Jeremy T. Fineman, Seth Gilbert, and Calvin Newport. Smoothed analysis of information spreading in dynamic networks. In Christian Scheideler, editor, *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*, volume 246 of *LIPIcs*, pages 18:1–18:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.DISC.2022.18.

9 Stefan Dobrev and Imrich Vrto. Optimal broadcasting in hypercubes with dynamic faults. *Inf. Process. Lett.*, 71(2):81–85, 1999. doi:10.1016/S0020-0190(99)00093-9.

10 Stefan Dobrev and Imrich Vrto. Optimal broadcasting in tori with dynamic faults. *Parallel Process. Lett.*, 12(1):17–22, 2002. doi:10.1142/S0129626402000781.

11 Benjamin Doerr and Mahmoud Fouz. Asymptotically optimal randomized rumor spreading. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 502–513. Springer, 2011. doi:10.1007/978-3-642-22012-8_40.

12 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983. doi:10.1137/0212045.

13 Rick Durrett and Dong Yao. Susceptible–infected epidemics on evolving graphs. *Electronic Journal of Probability*, 27:1–66, 2022.

14 Chinmoy Dutta, Gopal Pandurangan, Rajmohan Rajaraman, Zhifeng Sun, and Emanuele Viola. On the complexity of information spreading in dynamic networks. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 717–736. SIAM, 2013. doi:10.1137/1.9781611973105.52.

15 Antoine El-Hayek, Monika Henzinger, and Stefan Schmid. Brief announcement: Broadcasting time in dynamic rooted trees is linear. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, 2022.

16 Antoine El-Hayek, Monika Henzinger, and Stefan Schmid. Asymptotically tight bounds on the time complexity of broadcast and its variants in dynamic networks. In *14th Innovations in Theoretical Computer Science (ITCS)*, 2023.

17 Faith Ellen, Barun Gorain, Avery Miller, and Andrzej Pelc. Constant-length labeling schemes for deterministic radio broadcast. *ACM Trans. Parallel Comput.*, 8(3):14:1–14:17, 2021. doi:10.1145/3470633.

18 Patrick T Eugster, Rachid Guerraoui, A-M Kermarrec, and Laurent Massoulié. Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67, 2004. doi:10.1109/MC.2004.1297243.

19 Pierre Fraigniaud and Emmanuel Lazard. Methods and problems of communication in usual networks. *Discret. Appl. Math.*, 53(1-3):79–133, 1994. doi:10.1016/0166-218X(94)90180-5.

20 Matthias Függer, Thomas Nowak, and Kyrill Winkler. On the radius of nonsplit graphs and information dissemination in dynamic networks. *Discret. Appl. Math.*, 282:257–264, 2020. `doi:10.1016/j.dam.2020.02.013`.

21 Hugo Rincon Galeana, Ami Paz, Stefan Schmid, Ulrich Schmid, and Kyrill Winkler. The time complexity of consensus under oblivious message adversaries. In *14th Innovations in Theoretical Computer Science (ITCS)*, 2023.

22 Hugo Rincon Galeana, Ulrich Schmid, Kyrill Winkler, Ami Paz, and Stefan Schmid. Topological characterization of consensus solvability in directed dynamic networks. *arXiv preprint arXiv:2304.02316*, 2023. `doi:10.48550/arXiv.2304.02316`.

23 Mohsen Ghaffari, Fabian Kuhn, and Hsin-Hao Su. Distributed MST and routing in almost mixing time. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 131–140. ACM, 2017. `doi:10.1145/3087801.3087827`.

24 Juraj Hromkovič, Ralf Klasing, Burkhard Monien, and Regine Peine. Dissemination of information in interconnection networks (broadcasting & gossiping). In *Combinatorial network theory*, pages 125–212. Springer, 1996.

25 Fabian Kuhn, Nancy A. Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In Leonard J. Schulman, editor, *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 513–522. ACM, 2010. `doi:10.1145/1806689.1806760`.

26 Linyuan Lu, Austin Mohr, and László Székely. Quest for negative dependency graphs. In *Recent Advances in Harmonic Analysis and Applications*, pages 243–258. Springer, 2012.

27 Uri Meir, Ami Paz, and Gregory Schwartzman. Models of smoothing in dynamic networks. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 36:1–36:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPICS.DISC.2020.36`.

28 James D Murray et al. Mathematical biology i: an introduction, 2002.

29 Jim Pitman. Coalescent random forests. *Journal of Combinatorial Theory, Series A*, 85(2):165–193, 1999.

30 Martin Zeiner, Manfred Schwarz, and Ulrich Schmid. On linear-time data dissemination in dynamic rooted trees. *Discret. Appl. Math.*, 255:307–319, 2019. `doi:10.1016/j.dam.2018.08.015`.

On the Power of Graphical Reconfigurable Circuits

Yuval Emek
Technion - Israel Institute of Technology, Haifa, Israel

Yuval Gil
Technion - Israel Institute of Technology, Haifa, Israel

Noga Harlev
Technion - Israel Institute of Technology, Haifa, Israel

Abstract

We introduce the *graphical reconfigurable circuits (GRC)* model as an abstraction for distributed graph algorithms whose communication scheme is based on local mechanisms that collectively construct long-range reconfigurable channels (this is an extension to general graphs of a distributed computational model recently introduced by Feldmann et al. (JCB 2022) for hexagonal grids). The crux of the GRC model lies in its modest assumptions: (1) the individual nodes are computationally weak, with state space bounded independently of any global graph parameter; and (2) the reconfigurable communication channels are highly restrictive, only carrying information-less signals (a.k.a. *beeps*). Despite these modest assumptions, we prove that GRC algorithms can solve many important distributed tasks efficiently, i.e., in polylogarithmic time. On the negative side, we establish various runtime lower bounds, proving that for other tasks, GRC algorithms (if they exist) are doomed to be slow.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases graphical reconfigurable circuits, bounded uniformity, beeping

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.22

Related Version *Full Version*: `https://arxiv.org/pdf/2408.10761` [6]

1 Introduction

The *reconfigurable circuits* model was introduced recently by Feldmann et al. [7] and studied further by Padalkin et al. [14, 13]. It extends the popular *geometric amoebot* model for (synchronous) distributed algorithms running in the hexagonal grid by providing them with an opportunity to form long-range communication channels. This is done by means of a distributed mechanism that allows each node to bind together a subset of its incident edges (which can be thought of as installing internal "wires" between the corresponding ports); the long-range channels, a.k.a. *circuits*, are then formed by taking the transitive closure of these local bindings (see Sec. 1.1 for details). The circuits serve as *beeping channels*, enabling their participating nodes to communicate via information-less signals. The crux of the model is that the distributed mechanism that controls the circuit formation is invoked in every round (of the synchronous execution) so that the circuits can be reconfigured.

In contrast to the original geometric amoebot model which is tailored specifically to planarly embedded (hexagonal) grids, the reconfigurable circuits model can be naturally generalized to arbitrary graph topologies. The starting point of the current paper is the formulation of such a generalization that we refer to as the *graphical reconfigurable circuits (GRC)* model (formally defined in Sec. 1.1).

An important feature of the GRC model is that it is *uniform*: the actions of each node v in the (general) communication graph G are dictated by a (possibly randomized) state machine whose description is fully determined by the degree of v (and the local input provided to v if there is such an input), independently of any global parameter of G [2]. A clear advantage of

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 22; pp. 22:1–22:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

uniform algorithms is that they can be deployed in a "one size fits all" fashion, without any global knowledge of the graph on which they run. We further require that the aforementioned state machines admit a finite description, which means, in particular, that the state space of the state machines are bounded independently of any global graph parameter. This requirement is an obvious necessary condition for practical implementations; we subsequently refer to uniform distributed algorithms subject to this requirement as *boundedly uniform*.

Combining the bounded uniformity with the light demands of the beeping communication scheme, demands which are known to be easy to meet in practice [4, 8], we conclude that the GRC model provides an abstraction for distributed (arbitrary topology) graph algorithms that can be implemented over devices with slim computation and communication capabilities. In particular, the GRC model may open the gate for a rigorous investigation of distributed algorithms operating in (natural or artificial) biological cellular networks whose communication mechanism is based on bioelectric signaling, known to be the basis for long range (low latency) communication in such networks.

The main technical contribution of this paper is the design of GRC algorithms for various classic distributed tasks that terminate in polylogarithmic time. Some of these tasks (e.g., the construction of a minimum spanning tree) are inherently global and are known to be subject to congestion bottlenecks, thus demonstrating that despite their limited computation and communication power, GRC algorithms can overcome both "locality" and "bandwidth" barriers. In fact, as far as we know, these are the first distributed algorithms that solve such tasks in polylogarithmic time under any boundedly uniform model.

While GRC algorithms can bypass the congestion bottlenecks of some distributed tasks, other tasks turn out to be much harder: We prove that under certain conditions, runtime lower bound constructions, developed originally for the CONGEST model [15], can be translated, almost directly, to the GRC model, thus establishing runtime lower bounds for a wide class of tasks.

1.1 The GRC Model

In the current section, we introduce the distributed computational model used throughout this paper, referred to as the *graphical reconfigurable circuits (GRC)* model. A GRC algorithm `Alg` runs over a (finite simple) undirected graph $G = (V, E)$ so that each node $v \in V$ is associated with its own copy of a (possibly randomized) state machine defined by `Alg`; for clarity of the exposition, we often address node v and the state machine that dictates v's actions as the same entity (our intention will be clear from the context).

We adopt the *port numbering* convention [2, 10] stating that from the perspective of a node $v \in V$, each edge $e \in E(v)$ is identified by a unique port number taken from the set $\{1, \ldots, \deg(v)\}$.[1] Every edge $e \in E$ is associated with k *pins*, where $k \in \mathbb{Z}_{>0}$ is a constant determined by the algorithm designer;[2] these pins are represented as pairs of the form $p = (e, i)$ for $i \in [k]$. Let $P = E \times [k]$ denote the set of all pins. For a node $v \in V$, let $P(v) = E(v) \times [k]$ denote the set of pins associated with the edges incident on v. The GRC model is defined so that for each pin $p = (e, i) \in P(v)$, node v is aware of the (local) port

[1] Given an edge subset $F \subseteq E$ and a node $v \in V$, we denote the set of edges in F incident on v by $F(v) = \{e \in F \mid e \ni v\}$ and the degree of v by $\deg(v) = |E(v)|$.

[2] For the (asymptotic) upper bounds established in the current paper, it is actually sufficient to use $k = 1$ pins per edge. However, this is not true in general (see, e.g., [7, Sections 3.4 and 4.4]) and regardless, using multiple (yet, $O(1)$) pins per edge often facilitates the algorithm's exposition. In any case, we do not make an effort to optimize the value of k.

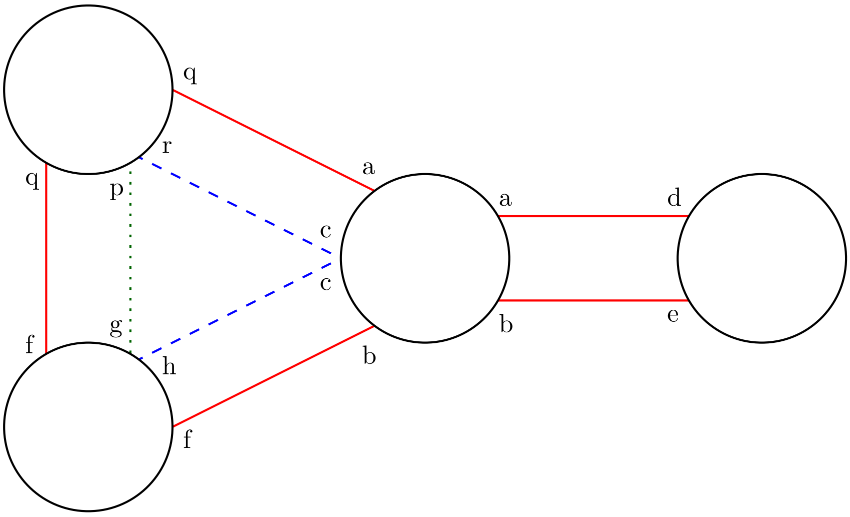


Figure 1 The circuits formed on a communication graph by the local node decisions. The graph includes 4 nodes, depicted by the black cycles, and 4 edges (not shown explicitly in the figure), each one of them is associated with $k = 2$ pins, depicted by the straight lines. The local pin partitions are presented by the lower-case letters. These local pin partitions result in forming three circuits, consisting of the red (solid) pins, the blue (dashed) pins, and the green (dotted) pin.

number of edge e as well as the (global) index $i \in [k]$. In particular, the other endpoint of edge e agrees with v on the index i of pin p although the two nodes may identify e by different port numbers.

The execution of algorithm `Alg` advances in synchronous *rounds*. Each round $t = 0, 1, \ldots$ is associated with a partition $\mathcal{C}^t$ of the pin set P into non-empty pairwise disjoint parts, called *circuits*. The partition $\mathcal{C}^0$ is defined so that each pin forms its own singleton circuit; for $t \geq 1$, the partition $\mathcal{C}^t$ is determined by the nodes according to a distributed mechanism explained soon.

For a round $t \geq 0$, a node $v \in V$ is said to *partake* in a circuit $C \in \mathcal{C}^t$ if $P(v) \cap C \neq \emptyset$. Let $\mathcal{C}^t(v) = \{C \in \mathcal{C}^t \mid P(v) \cap C \neq \emptyset\}$ denote the set of circuits in which node v partakes.

The communication scheme of the GRC model is defined on top of the circuits so that each circuit $C \in \mathcal{C}^t$ serves (during round t) as a *beeping channel* [4] for the nodes that partake in C. Before getting into the specifics of this communication scheme, let us explain how the partition $\mathcal{C}^t$ is formed based on the actions of the nodes in round $t - 1$.

Fix some round $t \geq 1$. Towards the end of round $t - 1$, each node $v \in V$ decides on a partition $\mathcal{R}^t(v)$ of $P(v)$, referred to as the *local pin partition* of v. Let $\mathcal{L}^t$ be the symmetric binary relation over P defined so that pins $p = (e, i)$ and $p' = (e', i')$ are related under $\mathcal{L}^t$ (i.e., $(p, p'), (p', p) \in \mathcal{L}^t$) if and only if there exists a node $v \in V$ (incident on both e and e') such that p and p' belong to the same part of $\mathcal{R}^t(v)$. Let $\mathrm{tc}(\mathcal{L}^t)$ be the reflexive transitive closure of $\mathcal{L}^t$, which is, by definition, an equivalence relation over P. The circuits in $\mathcal{C}^t$ are taken to be the equivalence classes of $\mathrm{tc}(\mathcal{L}^t)$. See Figure 1 for an illustration.[3]

[3] As presented by Feldmann et al. [7], the physical interpretation of the abstract circuit forming process is that each node v internally "wires" all pins belonging to the same part $R \in \mathcal{R}^t(v)$ to each other, thus ensuring that a signal transmitted over one pin in R is disseminated to all pins in R (and through them, to the entire circuit that contains R).

We are now ready to formally define the operation of each node $v \in V$ in round $t = 0, 1, \ldots$ This includes the following three steps, where we denote the state of v in round t by $S^t(v)$:
(1) Node v decides (possibly in a probabilistic fashion), based on $S^t(v)$, on a pin subset $B^t(v) \subseteq P(v)$ and *beeps* – namely, emits an information-less signal – on every pin in $B^t(v)$; we say that v *beeps* on a circuit $C \in \mathcal{C}^t(v)$ if v beeps on (at least) one of the pins in C.
(2) For each pin $p \in P(v)$, node v obtains a bit of information revealing whether at least one node beeps (in the current round) on the (unique) circuit $C \in \mathcal{C}^t$ to which p belongs.
(3) Node v decides (possibly in a probabilistic fashion), based on $S^t(v)$ and the information obtained in step (2), on the next state $S^{t+1}(v)$ and the next local pin partition $\mathcal{R}^{t+1}(v)$.
We emphasize that for each circuit $C \in \mathcal{C}^t(v)$ and pin $p \in P(v) \cap C$, node v can distinguish, based on the information obtained in step (2) for p, between the scenario in which zero nodes beep on C and the scenario in which a positive number of nodes beep on C, however, node v cannot tell how large this positive number is. In fact, if v itself decides (in step (1)) to beep on pin p, then v does not obtain any meaningful information from p in step (2) (in the beeping model terminology [1], this is referred to as lacking "sender collision detection").[4]

An important feature of the GRC model is that `Alg` is required to be *boundedly uniform*, namely, the number of states in the state machine associated with a node $v \in V$, as well as the description of the transition functions that determine the next state $S^{t+1}(v)$ and the next local pin partition $\mathcal{R}^{t+1}(v)$, are finite and fully determined by the local parameters of v, independently of any global parameter of the graph G on which `Alg` runs. These local parameters include the degree $\deg(v)$ of v and, depending on the specific task, any local input provided to v at the beginning of the execution (e.g., the weights of the edges incident on v).[5] In particular, node v does not "know" (and generally, cannot encode) the number $n = |V|$ of nodes, the number $m = |E|$ of edges, the maximum degree $\Delta = \max_{v \in V} \deg(v)$, or the diameter $D = \max_{u,v \in V} d_G(u, v)$.[6] Notice that the uniformity in n means that the nodes are also *anonymous*, i.e., they do not (and cannot) have unique identifiers.

The primary performance measure applied to our algorithms is their *runtime* defined to be the number of rounds until termination. When the algorithm is randomized, its runtime may be a random variable, in which case we aim towards bounding it whp.[7]

Relation to CONGEST. An adversity faced by GRC algorithms is the limited amount of information that can be sent/received by each node in a single round. Such limitations lie at the heart of the popular *CONGEST* [15] model that operates in synchronous message passing rounds, using messages of size B, where the typical choices for B are $B = O(1)$, $B = \Theta(\log n)$, or $B = \text{polylog}(n)$ (by definition, the uniform version of CONGEST adopts the former choice). An important point of similarity between the two models is that per round, both CONGEST and GRC algorithms can communicate $\tilde{O}(s)$ bits of information over a cut of

[4] The reader may wonder why the decisions made in step (1) and the information obtained in step (2) are centered on the pins in $P(v)$, rather than on the circuits in $\mathcal{C}^t(v)$. The reason is that node v is not necessarily aware of the partition induced on $P(v)$ by $\mathcal{C}^t(v)$ (i.e., the exact assignment of the pins in $P(v)$ to the circuits in $\mathcal{C}^t(v)$); indeed, the latter partition depends on the local pin partitions $\mathcal{R}^t(u)$ of other nodes $u \in V$, some of which may be far away from v. For example, in Figure 1, the local pin partition of the rightmost node separates between its two incident pins; nevertheless, both pins belong to the same (red) circuit due to local pin partitions decided upon in the other side of the graph.

[5] To maintain strict uniformity, we adhere to the convention that numerical values included in the local inputs (e.g., edge weights) are encoded as bitstrings without "leading zeros", thus ensuring that the length of such a bitstring by itself does not reveal any global information.

[6] The notation $d_G(u, v)$ denotes the distance (in hops) between nodes u and v in G.

[7] An event A holds *with high probability (whp)* if $\mathbb{P}(A) \geq 1 - n^{-c}$ for an arbitrarily large constant c.

Table 1 Our runtime upper bounds. The corresponding GRC algorithms are randomized and their correctness and runtime guarantees hold whp; the one exception is the spanner construction, where the number of edges is bounded in expectation.

task		runtime
construction	minimum spanning tree (integral edge weights $\in [1, W]$)	$O(\log(n) \cdot \log(n + W))$
	$(2\kappa - 1)$-spanner with $O(n^{1+(1+\varepsilon)/\kappa})$ edges in expectation	$O(\kappa + \log n)$
verification	minimum spanning tree (integral edge weights $\in [1, W]$)	$O(\log(n) \cdot \log(n + W))$
	simple path, connectivity, (s, t)-connectivity, connected spanning subgraph, cut, (s, t)-cut, Hamiltonian cycle, e-cycle containment, edge on all (s, t)-paths	$O(\log n)$

size s.[8] As explained in Sec. 3, from the perspective of message exchange per se (regardless of local computation), T CONGEST rounds can be simulated by $O(\log n + T \cdot B)$ GRC rounds whp, so, ignoring the additive logarithmic term, GRC algorithms are at least as strong as the boundedly uniform version of CONGEST algorithms. In fact, they are strictly stronger: the crux of GRC algorithms is that they enjoy the advantage of reconfigurable long-range communication channels (though highly restrictive ones); this advantage materializes in some of the GRC algorithms developed in the sequel whose runtime is significantly smaller than their corresponding (not necessarily uniform) CONGEST lower bounds.

1.2 Our Contribution

The main takeaway from this paper is that many important distributed tasks admit highly efficient GRC algorithms – see Table 1. Notice that with the exception of the sparse spanner construction, all tasks mentioned in Table 1 admit $\tilde{\Omega}(\sqrt{n} + D)$ runtime lower bounds under the (not necessarily uniform) CONGEST model [17, 16], demonstrating that reconfigurable beeping channels are a powerful tool even for boundedly uniform algorithms.

The polylogarithmic runtime upper bounds presented in Table 1 imply that the $\tilde{\Omega}(\sqrt{n}+D)$ CONGEST lower bounds for the corresponding tasks fail to transfer to the GRC model (refer to the full version [6] for further discussion of this "failed transfer"). CONGEST lower bounds for other distributed tasks on the other hand do transfer, almost directly, to GRC. Indeed, we develop a generic translation, from CONGEST runtime lower bounds to GRC runtime lower bounds, which applies to a large class of CONGEST lower bound constructions.

1.3 Paper's Outline

The remainder of this paper is organized as follows. We start in Sec. 2 with a discussion of the main technical challenges encountered towards establishing our results and the ideas used to overcome them. Sec. 3 introduces some preliminary definitions, as well as several basic procedures used in the later technical sections. The GRC algorithms promised in Table 1 for the tasks of constructing a minimum spanning tree and a spanner are presented and analyzed in Sec. 4 and 5, respectively. (Throughout, missing proofs and constructions are deferred to the full version [6].)

[8] The asymptotic notations $\tilde{O}(\cdot)$ and $\tilde{\Omega}(\cdot)$ hide polylog(n) expressions.

2 Technical Overview

In this section, we discuss the different challenges that arise in our constructions and present a brief overview of the technical ideas used to overcome these challenges; see Sec. 4 and 5 for the full details.

Minimum Spanning Tree. The minimum spanning tree (MST) construction follows the structure of Boruvka's classic algorithm [3]. The algorithm maintains a partition of the node set into *clusters* that correspond to the connected components of the subgraph induced by the edges which were already selected for the MST. It operates in phases, where the main algorithmic task in a phase is to identify a *lightest outgoing edge* for each cluster. The clusters are then merged over the identified edges, adding those edges to the output edge set.

If the edge weights are distinct, then no cycles are formed by the cluster merging process and Boruvka's algorithm is guaranteed to return an MST of the original graph. This well known fact is utilized by the existing distributed implementations of Boruvka's algorithm that typically use the unique node IDs to "enforce" distinct edge weights.

Unfortunately, obtaining distinct edge weights under our boundedly uniform model is hopeless. This means that the set L of lightest outgoing edges (of all clusters) cannot be safely added to the output edge set without the risk of forming cycles, thus forcing us to come up with an alternative mechanism. The key technical idea here is a procedure that runs in each phase independently and constructs (whp) a *total order* $\mathcal{T}$ over the set L. Following that, we identify a $\mathcal{T}$-minimal outgoing edge for each cluster and perform the cluster merger over the identified edges. As we prove in Sec. 4, selecting the $\mathcal{T}$-minimal outgoing edges ensures that no cycles are formed, resulting in a valid MST. Notice that for this argument to work, it is crucial that $\mathcal{T}$ is defined *globally* over all edges in L which is ensured by a careful design of the aforementioned procedure.

Spanner. The spanner construction is based on the elegant random shifts method of [12]. Particularly, the idea is similar to the distributed algorithm of [9] that uses random shifts to obtain a $(2\kappa - 1)$-spanner of expected size $O(n^{1+1/\kappa})$. The heart of the random shift method is a probabilistic clustering process based on a random variable δ_v drawn independently by each node $v \in V$. Specifically, in [9], each node $v \in V$ samples δ_v from the capped geometric distribution (see Sec. 3 for a definition) with parameters $p = 1 - n^{-1/\kappa}$ and $r = \kappa - 1$. The main challenge of adapting the algorithm to the boundedly uniform GRC model lies in the fact that the nodes are unable to sample from a distribution whose parameters depend on n. Nevertheless, we present a sampling procedure that allows each node $v \in V$ to sample δ_v from a distribution that is *sufficiently close* to the aforementioned capped geometric distribution.

As we prove in Sec. 5, the sampling procedure allows us to construct a spanner with nearly the same properties as those of [9]. More concretely, we extend and adapt the analysis of [9] to show that our algorithm constructs a spanner with stretch $2\kappa - 1$ whp, and size $O(n^{1+(1+\varepsilon)/\kappa})$ in expectation, where $\varepsilon > 0$ is a constant parameter that can be made desirably small.

3 Preliminaries

Graph Theoretic Definitions. Consider a connected graph $G = (V, E)$. Given an edge-weight function $w : E \to \mathbb{R}$, a *minimum spanning tree (MST)* of G with respect to w is an edge subset $T \subseteq E$ such that (V, T) is a spanning tree of G that minimizes the weight $w(T) = \sum_{e \in T} w(e)$.

For an edge subset $H \subseteq E$, let $d_H(u, v)$ denote the distance in the graph (V, H) between two nodes $u, v \in V$. For an integer $\sigma > 0$, we say that $H \subseteq E$ is a *σ-spanner* of G if $d_H(u, v) \leq \sigma \cdot d_G(u, v)$ for all $u, v \in V$. Equivalently, H is a σ-spanner if and only if $d_H(u, v) \leq \sigma$ for every edge $(u, v) \in E$. The *stretch* of H is defined as the smallest value σ for which H is a σ-spanner.

The parts of a partition $\mathcal{P}$ of the node set V are often referred to as *clusters*. We say that clusters S and S', $S \neq S'$, are *neighboring clusters* if there exists an edge $(v, v') \in E$ such that $v \in S$ and $v' \in S'$. In this case, we say that edge (v, v') *bridges* the clusters S and S', and more broadly, refer to (v, v') as a *bridging* edge of $\mathcal{P}$. We say that an edge $(u, v) \in E$ is an *outgoing* edge of cluster S if $u \in S$ and $v \notin S$. For a cluster S, let $\partial_S \subseteq E$ denote the set of edges outgoing from S.

Capped Geometric Distribution. For parameters $p \in [0, 1]$ and $r \in \mathbb{Z}_{>0}$, the *capped geometric distribution*, denoted by $GeomCap(p, r)$, is defined by taking $\mathbb{P}[GeomCap(p, r) = i]$ to be $p(1-p)^i$ if $i \in \{0, \ldots, r-1\}$; $(1-p)^r$ if $i = r$; and 0 otherwise. Intuitively, the distribution relates to r Bernoulli experiments indexed by $0, \ldots, r-1$, each with success probability p. A random variable sampled from the capped geometric distribution represents the index of the first successful experiment, whereas it is equal to r if all experiments fail. The capped geometric distribution admits a memoryless property for the values $0 \leq i \leq r-1$. In particular, a useful identity that follows is $\mathbb{P}[X = i \mid X \geq i] = \mathbb{P}[X = 0] = p$ for a random variable $X \sim GeomCap(p, r)$ and an index $0 \leq i \leq r-1$.

3.1 Auxiliary Procedures

Global Circuits. The algorithms presented in this paper utilize a *global circuit*, i.e., a circuit in which every node $v \in V$ partakes. A global circuit can be constructed in round $t \geq 0$ as follows. For some index $1 \leq i \leq k$, every node $v \in V$ partitions its pin set in round t such that $E(v) \times \{i\} \in \mathcal{R}^t(v)$.

Procedure `CountingToLogn`. We next present a procedure referred to as `CountingToLogn`, whose runtime is $\Theta(\log n)$ rounds whp. While the uniformity in n prevents the nodes from counting $\log n$ rounds individually, the duration of this procedure can indicate to the nodes that whp, $\Theta(\log n)$ rounds have passed. The nodes first construct a global circuit, as described above. Throughout the procedure, the nodes maintain a node set $M \subseteq V$ of *competitors*, where initially $M = V$. In each round, each competitor $v \in M$ tosses a fair coin and beeps through the global circuit if the coin lands heads. If the coin lands tails, v removes itself from M. The procedure terminates when no competitor beeps through the global circuit.

We show the following useful property regarding the runtime of the described procedure.

▶ **Lemma 3.1.** *For an integer $r > 0$, consider $2r - 1$ independent executions of* `CountingToLogn` *and let τ be the median runtime of these executions (i.e., the r-th fastest runtime). For any constant $0 < \rho < 1$, it holds that $\mathbb{P}[(1-\rho)\log n \leq \tau \leq (1+\rho)\log n] \geq 1 - 2n^{-\rho r}$.*

Simulating a Message-Passing Network. In a *message-passing* network, in each round, every pair of neighboring nodes may exchange single bit messages with each other (cf. the CONGEST(1) model [15]). One can simulate a message-passing network in the GRC model using relatively standard techniques as cast in the following theorem.

▶ **Theorem 3.2.** *Let* `Alg` *be a GRC algorithm where additionally, in each round, each node is able to exchange* 1*-bit messages with its neighbors. If the runtime of* `Alg` *is* T*, then it can be transformed into an algorithm* `Alg′` *in the GRC model (without messages between neighbors) with a runtime of* $O(\log n) + 4T$ *whp.*

For simplicity of presentation, we subsequently utilize Thm. 3.2 and describe our algorithms as if the nodes can exchange 1-bit messages with their neighbors in each round.

Leader Election. In the leader election task, the goal is for a single node in a given node set $I \subseteq V$ to be selected as a *leader*, whereas all other nodes of I are selected to be *non-leaders*. Leader election is used as a procedure in some of our algorithms. To that end, we use a leader election algorithm presented by Feldmann et al. [7] in the context of reconfigurable circuits in the geometric amoebot model. We note that this leader election algorithm only uses a global circuit (as described above) and thus can be applied as-is in the GRC model. Hence, the following theorem is established.

▶ **Theorem 3.3** ([7])**.** *The leader election task can be solved within* $O(\log n)$ *rounds whp.*

Outgoing Edge Detection. Consider a graph $G = (V, E)$ and let $H \subseteq E$ be a subset of edges such that each node $v \in V$ knows the set of incident edges $H(v)$. Define a partition of V into clusters according to the connected components of (V, H). The objective of this procedure is for each node $v \in V$ to determine for each neighbor $u \in N(v)$, whether u belongs to the same cluster as v. To that end, the nodes first construct a circuit for each cluster. This is done by each node $v \in V$ including the pin subset $H(v) \times \{i\}$ as part of its local pin partition for some $i \in [k]$ (i is the same for all nodes). Then, each cluster elects a leader utilizing the leader election algorithm mentioned above. The selected leader of each cluster tosses $\Theta(\log n)$ bits and beeps them through the cluster's circuit, one at a time (a beep represents 1 and silence represents 0). Since the nodes cannot count $\Theta(\log n)$ rounds, Proc. `CountingToLogn` is executed in parallel through a global circuit for (a sufficiently large) $c > 1$ times, indicating to the clusters' leaders how long to continue with the bit tossing process. Every node $v \in V$ sends every bit received through its cluster's circuit in a direct message to all its neighbors (messages between neighbors are executed by means of the simulation method described in Sec. 3.1). For every incident edge $e \in E(v)$, node v checks if the bit received differs from the bit sent. If so, e is classified by v as an outgoing edge.

▶ **Lemma 3.4.** *In the outgoing edge detection procedure, every edge* $e = (u, v) \in E$ *is classified correctly whp by both* u *and* v*.*

▶ **Lemma 3.5.** *The outgoing edge detection procedure takes* $\Theta(\log n)$ *rounds whp.*

4 A Fast Minimum Spanning Tree Algorithm

In this section, we present a randomized MST algorithm that operates in the GRC model. As common in the distributed setting, we assume the edge-weights are integers from the set $\{1, \ldots, W\}$ for some positive integer W. Each node $v \in V$ initially knows only the weights of edges in $E(v)$. In particular, as dictated by the GRC model, node v does not know the value of W or any other information about W.

Our algorithm can be seen as an adaptation of Boruvka's classical MST algorithm [3] to the GRC model. Throughout its execution, Boruvka's algorithm maintains an edge set T and a cluster partition defined such that each cluster is a connected component of (V, T).

Initially, $T = \emptyset$ (and each node is a cluster). At each iteration of the algorithm, each cluster S adds a lightest outgoing edge $e^* = \arg\min_{e \in \partial_S}\{w(e)\}$ to T. This means that S merges with the neighboring cluster S' that is incident on e^*. It is well-known that if the edge weights are unique, then Boruvka's algorithm computes an MST of G. Notice that in our case, edge weights are not necessarily unique, so we construct a symmetry-breaking mechanism based on a total order of the lightest outgoing edges as explained later on.

The algorithm begins with an empty set of *tree edges* and operates in phases. The goal of each phase is to add tree edges similarly to Boruvka's algorithm. Let $T_i \subseteq E$ denote the tree edges at the end of phase $i \geq 0$. As in Boruvka's algorithm, the connected components of (V, T_i) are defined to be the *clusters* at the beginning of phase $i + 1$. The nodes construct a designated circuit for each cluster formed during the algorithm. Additionally, the nodes communicate through a global circuit and exchange messages with their neighbors using the methods described in Sec. 3. The operation of each phase is divided into the following stages.

Outgoing Edge Detection. The purpose of this stage is to allow the nodes to identify which of their incident edges is an outgoing edge. To that end, the nodes execute the outgoing edge detection procedure described in Sec. 3.1. When the procedure terminates, each node detecting an outgoing edge beeps through the global circuit. The algorithm terminates if no node beeps in this round through the global circuit. Otherwise, the nodes advance to the next stage. Denote by $\texttt{Out}(v)$ the set of edges classified as outgoing by node $v \in V$.

Lightest Edge Detection. In this stage, each cluster searches for its lightest outgoing edges. Fix some cluster S. At the beginning of this stage, every node $v \in S$ such that $\texttt{Out}(v) \neq \emptyset$ marks a single edge $e \in \texttt{Out}(v)$ with weight $w(e) = \min_{e' \in \texttt{Out}(v)} w(e')$ as a candidate. The comparison between weights of the candidate edges incident on the nodes of S is done in two steps.

First, the nodes compare the lengths of the candidate edge weights (i.e., the number of bits in the edge-weight representation). Consider a node $v \in S$ incident on a candidate edge e, and let $\ell_v = \lfloor \log w(e) \rfloor + 1$ be the length of $w(e)$. Node v counts $\ell_v - 1$ rounds. If v hears a beep on the cluster's circuit during those $\ell_v - 1$ rounds, then v unmarks e as a candidate. Otherwise, v beeps through the cluster's circuit in round ℓ_v and keeps e as a candidate edge. Following the first step, all remaining candidate edges of S have weights of the same length. In the second step, the weights of the candidate edges of S are compared bit by bit, starting from the most significant bit. Let $v \in S$ be a node that still has an incident candidate edge e. The second step runs for ℓ_v rounds indexed by $j = 1, \dots, \ell_v$. In round j, if e is still a candidate, then v beeps through the cluster's circuit if and only if the j-th most significant bit of $w(e)$ is 0. If v did not beep but heard a beep through the cluster's circuit, it unmarks e as a candidate edge. Notice that at the end of the second step, only the lightest edges that were classified as outgoing remain candidates.

In parallel, v beeps through the global circuit at every round of the stage in which e is still a candidate. Once v finishes the stage (either because e was marked as a lightest outgoing edge or e was unmarked as a candidate), it stops beeping through the global circuit. The stage terminates when no beep is transmitted through the global circuit.

Single Edge Selection. At this point, only the edges marked as lightest outgoing edges of each cluster remained candidates. However, there may be more than one candidate edge for some clusters. The goal of this stage is to select a single edge for each cluster while avoiding the formation of a cycle in the output edge set (as we will show in the analysis). To that end, every node $v \in V$ with an incident candidate edge (u, v) informs u that (u, v) is still a candidate. Then, each of u and v draws a random bit denoted by $u.bit$ and $v.bit$, respectively. Node u sends $u.bit$ to v and v calculates the bitwise XOR of $u.bit$ and $v.bit$. Node v beeps through the cluster's circuit if the XOR result is 1. If node v does not beep for edge e but hears a beep through the cluster's circuit, it unmarks e as a candidate. Notice that if (u, v) is lightest with regard to u's cluster as well, then the same operation is performed also by u using the same drawn bits. This edge selection process is done in parallel to Proc. `CountingToLogn` over the global circuit, executed (a sufficiently large) $c > 1$ times. The nodes continue to draw bits for their incident candidate edges as long as Proc. `CountingToLogn` continues. If a node $v \in V$ has an incident candidate edge $e = (u, v)$ at the end of this stage, then it informs u, and both endpoints mark e as a tree edge.

Updating the Local Pin Partition. Every node $v \in V$ sets its local pin partition to include the pin subset $T(v) \times \{j\}$ for some $j \in [k]$, where $T(v)$ is the set of edges incident on v that were marked as tree edges (either in the current or a prior phase). Observe that this local pin partition by the nodes constructs a circuit for every cluster.

The output of the algorithm is the set of all tree edges.

4.1 Analysis

In this section, we prove the correctness and analyze the runtime of the MST algorithm presented above, establishing the following theorem.

▶ **Theorem 4.1.** *The algorithm constructs an MST of G whp and runs in $O(\log n \cdot \log(n+W))$ rounds whp.*

Recall that $T_i \subseteq E$ is the set of tree edges at the end of phase $i = 0, 1, \ldots$ and let i^* be the last phase of the algorithm. Let q_i be the number of clusters maintained by the algorithm at the beginning of phase i, that is, the number of connected components in (V, T_i).

▶ **Lemma 4.2.** *Consider a phase $0 \leq i \leq i^*$. If $q_i = 1$, then the algorithm terminates in phase i whp; otherwise, $q_{i+1} \leq \frac{1}{2} q_i$ whp.*

Notice that since the algorithm starts with n clusters, Lem. 4.2 implies the following corollary.

▶ **Corollary 4.3.** *The algorithm terminates after $i^* = O(\log n)$ phases whp. Moreover, the subgraph (V, T_{i^*}) is connected whp.*

Denote by $D_i \subseteq E$ the set of edges that are candidates for some (at least one) cluster at the end of the single edge selection stage of phase i (to be marked as tree edges).

▶ **Lemma 4.4.** *The subgraph (V, T_{i^*}) is a spanning tree of G whp.*

Proof. By Cor. 4.3, (V, T_{i^*}) is connected whp. So, it is left to show that (V, T_{i^*}) is a forest whp. We prove by induction over the phases that (V, T_i) is a forest whp for all $0 \leq i \leq i^*$. Cor. 4.3 also guarantees that there are $O(\log n)$ phases whp; hence the statement follows by applying union bound over the phases.

For the base of the induction, notice that $T_0 = \emptyset$, and thus (V, T_0) is a forest. Now, suppose that (V, T_i) is a forest for some $0 \leq i < i^*$. We show that $(V, T_{i+1}) = (V, T_i \cup D_i)$ is a forest whp. For every edge $e \in D_i$, let $B_i(e)$ be the integer obtained from the binary representation of the bit sequence drawn for e by its endpoints (i.e., the sequence of XORed bits) in the single edge selection stage of phase i. Define the binary relation $\prec_i$ for every two edges $e, e' \in D_i$ as:

$$e \prec_i e' \iff w(e) < w(e') \vee (w(e) = w(e') \wedge B_i(e) > B_i(e')) \ .$$

Notice that by repeating the `CountingToLogn` for a sufficiently large number of times, we get that the $B_i(\cdot)$ values are unique whp. By the construction of the single edge selection stage, this means that each cluster selects exactly one outgoing edge whp – the lightest outgoing edge which is minimal with respect to $\prec_i$. To complete our proof, we show that if the $B_i(\cdot)$ values are unique and (V, T_i) is a forest, then $(V, T_{i+1}) = (V, T_i \cup D_i)$ is a forest.

Assume by contradiction that there exists at least one cycle in $(V, T_i \cup D_i)$ and let Y be a simple cycle in $(V, T_i \cup D_i)$. By the induction hypothesis we know that (V, T_i) is a forest, therefore $Y \cap D_i \neq \emptyset$. Let $e \in Y \cap D_i$ be the (unique) largest edge (with respect to $\prec_i$) of $Y \cap D_i$, and let S be the cluster that selected e. Observe that since (V, T_i) is a forest and Y is a cycle, there exists another edge $e' \in Y \cap D_i - \{e\}$ which is an outgoing edge of S. However, by the choice of e, we know that $e' \prec_i e$, in contradiction to the selection of e by S. ◀

The following lemma asserts the correctness of our MST algorithm.

▶ **Lemma 4.5.** *The graph (V, T_{i^*}) is an MST of G whp.*

Proof. By Lem. 4.4, the graph (V, T_{i^*}) is a spanning tree of G whp. The proof of Lem. 4.4 shows that every cluster selects a single lightest outgoing edge in each phase whp. The statement then follows from the correctness of Boruvka's algorithm [3]. ◀

It remains to analyze the runtime of the algorithm.

▶ **Lemma 4.6.** *The MST algorithm runs in $O(\log n \cdot \log(n + W))$ rounds whp.*

Proof. By Corollary 4.3, the algorithm runs for $O(\log n)$ phases whp. We are left to bound the runtime of each phase. Every execution of the leader election algorithm and Proc. `CountingToLogn` takes $O(\log n)$ rounds whp. Hence, the outgoing edge detection and single edge selection stages each take $O(\log n)$ rounds whp. The lightest edge detection stage completes in $O(\log W)$ rounds, and updating the local pin partition does not require any communication. Therefore, every phase of the algorithm completes in $O(\log n + \log W)$ rounds whp. Overall, we get a runtime bound of $O(\log n(\log n + \log W)) = O(\log n \cdot \log(n \cdot W)) = O(\log n \cdot \log(n + W))$ rounds whp, where the last equality hods because $\log(n \cdot W) = \log n + \log W = O(\log(n + W))$. ◀

5 A Sparse Spanner Algorithm

In this section, we present a randomized spanner algorithm that operates in the GRC model. Given a parameter $\kappa \in \mathbb{Z}_{>0}$ and a constant $0 < \varepsilon < 1$, the algorithm constructs a spanner with a stretch of $(2\kappa - 1)$ whp and $O(n^{1+(1+\varepsilon)/\kappa})$ edges in expectation. More concretely, we prove the following theorem.

▶ **Theorem 5.1.** *There exists an algorithm in the GRC model that computes a set $H \subseteq E$ of edges such that H is a $(2\kappa - 1)$-spanner whp, and $\mathbb{E}[|H|] = O(n^{1+\frac{1+\varepsilon}{\kappa}})$. The runtime of the algorithm is $O(\kappa + \log n)$ rounds whp, and the memory space used by each node $v \in V$ is $O(\deg(v) + \kappa)$.*

In the full version [6], we present a modification of our algorithm to accommodate a memory space of only $O(\deg(v) + \log \kappa)$ for each node $v \in V$, at the cost of a slightly slower $O(\kappa \log n)$-round algorithm. We now describe the algorithm stated in Thm. 5.1.

The algorithm is based on the random shift concept introduced by Miller et al. in [12] and studied further in various works (see, e.g., [11, 5, 9]). We now give a high-level overview of a spanner construction algorithm based on the random shift approach (see [9] for the full details).

The algorithm starts with each node $v \in V$ sampling a value $\delta_v \sim \mathit{GeomCap}(1-n^{-1/\kappa}, \kappa-1)$ (see Sec. 3 for the capped geometric distribution definition). Then, the nodes conceptually add a virtual node s. Each node $v \in V$ adds an edge (s, v) of weight $w(s, v) = \kappa - \delta_v$ to form the graph G', where all other edges are assigned a unit weight. Following that, the nodes construct a shortest path tree T rooted at s. The nodes of G are partitioned into clusters defined by the connected components of T after removing s and its incident edges. To construct the spanner H, the nodes first add the (non-virtual) edges of T. Then, the nodes add edges to H such that for each edge $(u, v) \in E - T$, at least one of the following is satisfied: (1) H contains exactly one edge between u and a node in v's cluster; or (2) H contains exactly one edge between v and a node in u's cluster. As discussed in [9], the constructed edge-set H is a $(2\kappa - 1)$-spanner of expected size $O(n^{1+1/\kappa})$.

Our algorithm works in three stages as described below.

Sampling Procedure. Recall that the algorithm of [9] begins with each node $v \in V$ sampling $\delta_v \sim \mathit{GeomCap}(1 - n^{-1/\kappa}, \kappa - 1)$. Note that sampling from $\mathit{GeomCap}(1 - n^{-1/\kappa}, \kappa - 1)$ requires the nodes to know the value of n, which is not possible in the GRC model. Hence, we devise a designated sampling procedure for each node $v \in V$.

Let us first present the intuition behind the sampling procedure. The idea is for each node $v \in V$ to simulate $\kappa - 1$ *experiments*, each with success probability close to $1 - n^{-1/\kappa}$, and compute δ_v accordingly. To achieve such success probability without knowing n, Proc. `CountingToLogn` is utilized. In order to enhance the proximity to $1 - n^{-1/\kappa}$, Proc. `CountingToLogn` is executed numerous times in parallel, and δ_v is computed based on the run with median runtime.

For ease of presentation, we describe the sampling procedure in two stages. First, a sub-procedure referred to as the *basic* scheme is described. We later explain how this basic scheme is used in the sampling procedure. The basic scheme runs during an execution of Proc. `CountingToLogn`. For each node $v \in V$, let $b_v = (b_v[0], \ldots, b_v[\kappa - 2])$ be a vector of $\kappa - 1$ bits initialized to $b_v = (0, \ldots, 0)$. The purpose of entry $b_v[j]$ is to represent the success/failure of the i-th experiment for each $0 \leq j \leq \kappa - 2$. Let ε' be the largest value such that $1/(1 - \varepsilon')$ is an integer and $\varepsilon' \leq \varepsilon/(2 + \varepsilon)$. In each round j such that $j \bmod \kappa \neq 0$, each node v draws $1/(1 - \varepsilon')$ bits uniformly at random and sets $b_v[(j - 1) \bmod \kappa] = 1$ if any of those bits are 1.

In the sampling procedure, the nodes perform $c' = 2 \cdot \lceil c/\varepsilon' \rceil - 1$ executions of the basic scheme, where $c > 0$ is a constant. Let us index these executions by $i = 0, \ldots, c' - 1$. Starting from the execution indexed 0, the rounds of the executions are done alternately, i.e., a round of the run indexed by i is followed by a round of the run indexed by $(i + 1) \bmod c'$. Accordingly, each node $v \in V$ maintains c' vectors, $b_v^0, \ldots, b_v^{c'-1}$, each of size $\kappa - 1$ bits, such that b_v^i is the vector maintained by v during the i-th execution of the basic scheme. Additionally, v maintains a counter initialized to 0, whose goal is to count the executions that terminated. Whenever an execution terminates, the counter is increased by 1. Following the termination, during the rounds that are associated with that execution, the nodes do nothing. The nodes halt the executions when the counter reaches $\lceil c/\varepsilon' \rceil$ (notice that the counter is updated in the same manner for all nodes, thus they halt at the same time). Let $\tilde{i}$ denote the index of the execution in which the counter reached $\lceil c/\varepsilon' \rceil$. Observe that this is the $\lceil c/\varepsilon' \rceil$-th fastest execution, i.e., the execution with median runtime. Each node $v \in V$ defines δ_v to be the smallest index $0 \leq j \leq \kappa - 2$ for which $b_v^{\tilde{i}}[j] = 1$ if such an index exists, or $\delta_v = \kappa - 1$ otherwise.

Partition Into Clusters. Let G' be the graph formed by adding a virtual node s and edge (s, v) of weight $w(s, v) = \kappa - \delta_v$ for every $v \in V$. To compute the cluster partition, the nodes first construct a shortest path tree T rooted at s. The idea is simple: If $w(s, v) = 1$, then v sends a message to all its neighbors and marks itself as the center of its cluster. Otherwise, assume first that v receives a message in at least one of the rounds $2, \ldots, w(s, v) - 1$ and let $2 \leq i < w(s, v) - 1$ be the first such round. After receiving a message in round i, node v (arbitrarily) chooses a neighbor u that sent v a message in that round and adds the edge (u, v) into T. Then, in round $i + 1$, node v sends a message to all neighbors from which it did not receive a message in round i. Otherwise, if v does not receive a message after $w(s, v) - 1$ rounds, then in round $w(s, v)$ node v sends a message to all its neighbors and sets itself as the center of its cluster. Notice that after at most κ rounds, T is a shortest path tree rooted at s. The edges of T are added to the spanner H. The clusters are defined to be the connected components of (V, T) (i.e., the connected components formed by removing s and its incident edges). The nodes then construct a circuit for each cluster (similarly to the MST algorithm of Sec. 4). Observe that by design, each cluster has exactly one center. Note that every message sent in each round of this stage is of size one bit.

Addition of Bridging Edges. The construction of H is completed by the following procedure whose goal is to augment H with some of the edges that bridge between clusters. This is done by each cluster randomly drawing an ID. Then, each node $v \in V$ identifies its neighboring clusters with smaller IDs and adds a single edge to each such cluster into H.

Formally, each node $v \in V$ maintains a set $S^{\text{eq}}(v)$ initialized to be $N(v)$, and a set $S^{\text{sml}}(v)$ initialized to be $\emptyset$. Additionally, throughout the execution, v maintains a partition of $S^{\text{sml}}(v)$ into subsets according to the (randomly drawn) cluster IDs. The nodes engage in a process that runs in parallel to $4c + 7$ iterations of Proc. `CountingToLogn`. In each round of this process, every cluster center tosses a coin and communicates the outcome through the cluster's circuit to all the nodes in its cluster. Then, every node $v \in V$ sends a message with the coin toss received from its cluster's center to all neighbors. Let $S_i^{\text{eq}}(v)$ be the set $S^{\text{eq}}(v)$ at the beginning of round i. For each $u \in S_i^{\text{eq}}(v)$, if u and v sent the same bit, then u stays in $S^{\text{eq}}(v)$; otherwise, u is removed. Additionally, if u's bit is smaller than v's, then u is added to $S^{\text{sml}}(v)$. The partition of the nodes in $S^{\text{sml}}(v)$ is defined so that u and u' are in the same subset by the end of round i if and only if they were in the same subset at the beginning of round i and sent the same bit in round i. Let $s_1, \ldots, s_q$ be the partition of $S^{\text{sml}}(v)$ at the end of the process. For each $j \in [q]$, node v (arbitrarily) selects a single node $u \in s_j$ and adds the edge (u, v) into H.

This completes the construction of H. We now turn to analyzing the algorithm.

5.1 Analysis

This section is dedicated to proving Thm. 5.1. To that end, we start with a structural lemma about the capped geometric distribution.

▶ **Lemma 5.2.** *For arbitrary values $q_1, \dots, q_n$ and for $X_1, \dots, X_n \sim GeomCap(\phi, \kappa - 1)$, define $M = \max_{i \in [n]}\{X_i - q_i\}$. For the set $I = \{i \mid X_i < \kappa - 1 \ \wedge \ X_i - q_i \in \{M-1, M\}\}$, it holds that $\mathbb{E}[|I|] \leq \frac{2}{1-\phi}$.*

Recall that in the sampling procedure of our algorithm, the value δ_v is computed for each node $v \in V$ based on the $\tilde{i}$-th execution of the basic scheme, i.e., the execution that admits the median runtime. Particularly, within that execution, δ_v is defined as the first successful experiment out of $0, \dots, \kappa - 2$; or $\kappa - 1$ if all experiments failed. Let ϕ be the success probability of each such experiment and notice that ϕ itself is a random variable that depends on the execution's length. Define A to be the event that $1 - n^{-1/\kappa} \leq \phi \leq 1 - n^{-(1+\varepsilon)/\kappa}$. We prove the following lemma.

▶ **Lemma 5.3.** $\mathbb{P}[A] \geq 1 - 2n^{-c}$.

We now consider the bridging edges addition stage of the algorithm. Let B denote the event that for every edge $(u,v) \in E - T$, at least one of the following is satisfied: (1) H contains exactly one edge between u and a node in v's cluster; or (2) H contains exactly one edge between v and a node in u's cluster. We state the following.

▶ **Lemma 5.4.** $\mathbb{P}[B] \geq 1 - 3n^{-c}$.

For each node $v \in V$, let $M_v = \max_{u \in V}\{\delta_u - d_G(u,v)\}$ and $R(v) = \{u \in V \mid M_v - 1 \leq \delta_u - d_G(u,v) \leq M_v\}$. We obtain the following observation.

▶ **Observation 5.5.** *Consider an edge $(u,v) \in H$ such that u and v belong to clusters centered at nodes u' and v', respectively. Then, $u' \in R(v)$ or $v' \in R(u)$.*

We are now prepared to bound the expected number of edges in the spanner.

▶ **Lemma 5.6.** $\mathbb{E}[|H|] \leq 2n^{1+(1+\varepsilon)/\kappa} + n^{1+1/\kappa} + 1$.

Proof. By the law of total expectation,

$$\mathbb{E}[|H|] \ = \ \mathbb{E}[|H| \mid A \wedge B] \cdot \mathbb{P}[A \wedge B] + \mathbb{E}[|H| \mid \neg A \vee \neg B] \cdot \mathbb{P}[\neg A \vee \neg B]\,.$$

Combining Lem. 5.3 with Lem. 5.4, we get $\mathbb{P}[\neg A \vee \neg B] \leq 5n^{-c}$, and since $\mathbb{E}[|H| \mid \neg A \vee \neg B] \leq m < n^2$, it follows that

$$\mathbb{E}[|H|] \leq \mathbb{E}[|H| \mid A \wedge B] \cdot \mathbb{P}[A \wedge B] + n^2 \cdot 5n^{-c} \leq \mathbb{E}[|H| \mid A \wedge B] + 1\,,$$

where the final inequality holds for, e.g., $c \geq 3$. Therefore, we are left to bound the term $\mathbb{E}[|H| \mid A \wedge B]$.

Obs. 5.5 implies that the sum $\sum_{v \in V} |R(v)|$ accounts for every edge in H at least once, i.e., $\sum_{v \in V} |R(v)| \geq |H|$. Fix some node $v \in V$, we seek to bound $\mathbb{E}[|R(v)|]$. Partition the set $R(v)$ into $R_1(v) = \{u \in R(v) \mid \delta_u = \kappa - 1\}$ and $R_2(v) = R(v) - R_1(v)$. Notice that the events $\delta_u = \kappa - 1$ and B are independent. Thus, we get

$$\mathbb{E}[|R_1(v)| \mid A \wedge B] \leq n \cdot \mathbb{P}[\delta_u = \kappa - 1 \mid A \wedge B] \ = \ n \cdot \mathbb{P}[\delta_u = \kappa - 1 \mid A]\,.$$

Observe that $\mathbb{E}[|R_1(v)|] \leq n \cdot \mathbb{P}[\delta_u = \kappa - 1] = n(1-\phi)^{\kappa-1}$, and recall that if event A occurs, then $\phi \geq 1 - n^{-1/\kappa}$. Hence, it follows that

$$n \cdot \mathbb{P}[\delta_u = \kappa - 1 \mid A] \ = \ n(1-\phi)^{\kappa-1} \leq n \cdot n^{(-1/\kappa)\cdot(\kappa-1)} \ = \ n^{1/\kappa}\,.$$

As for R_2, applying Lem. 5.2, we get $\mathbb{E}[|R_2(v)|] \leq 2/(1-\phi)$. Once again, we condition on A and B to get

$$\mathbb{E}[|R_2(v)| \mid A \wedge B] = \mathbb{E}[|R_2(v)| \mid A] \leq 2/n^{-(1+\varepsilon)/\kappa} = 2n^{(1+\varepsilon)/\kappa} .$$

Overall, we conclude that

$$\begin{aligned}\mathbb{E}[|H|] &\leq n \cdot \mathbb{E}[R(v)] \leq n \cdot \mathbb{E}[|R_1(v)| \mid A \wedge B] + n \cdot \mathbb{E}[|R_2(v)| \mid A \wedge B] + 1 \\ &\leq 2n^{1+(1+\varepsilon)/\kappa} + n^{1+1/\kappa} + 1 .\end{aligned}$$ ◀

Next, we bound the stretch of H.

▶ **Lemma 5.7.** *H is a $(2\kappa-1)$-spanner whp.*

Proof. We now argue that if event B occurs, then H has stretch $2\kappa-1$, which implies the stated claim due to Lem. 5.4. To see that, consider an edge $(u,v) \in E$. Observe that the diameter within each cluster is at most $2\kappa-2$. This is because every node is at distance at most $\kappa-1$ from its cluster's center. Hence, if u and v belong to the same cluster, then $d_H(u,v) \leq 2\kappa-2$. Otherwise, if event B occurs, then either there is an edge $(\tilde{u},v) \in H$ between v and a node $\tilde{u}$ in u's cluster, or an edge $(u,\tilde{v}) \in H$ between u and a node $\tilde{v}$ in v's cluster. Assume w.l.o.g. that $(\tilde{u},v) \in H$. It follows that $d_H(u,v) \leq d_H(u,\tilde{u}) + d_H(\tilde{u},v) \leq 1 + 2\kappa - 2 = 2\kappa - 1$. ◀

This concludes the analysis of our algorithm.

References

1 Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. *Distributed Comput.*, 26(4):195–208, 2013. doi:10.1007/S00446-012-0175-7.

2 Dana Angluin. Local and global properties in networks of processors (extended abstract). In Raymond E. Miller, Seymour Ginsburg, Walter A. Burkhard, and Richard J. Lipton, editors, *Proceedings of the 12th Annual ACM Symposium on Theory of Computing (STOC)*, pages 82–93. ACM, 1980. doi:10.1145/800141.804655.

3 Otakar Boruvka. Contribution to the solution of a problem of economical construction of electrical networks. *Elektronický Obzor*, 15:153–154, 1926.

4 Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*, volume 6343 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2010. doi:10.1007/978-3-642-15763-9_15.

5 Michael Elkin and Ofer Neiman. Efficient algorithms for constructing very sparse spanners and emulators. *ACM Trans. Algorithms*, 15(1):4:1–4:29, 2019. doi:10.1145/3274651.

6 Yuval Emek, Yuval Gil, and Noga Harlev. On the power of graphical reconfigurable circuits, 2024. arXiv:2408.10761.

7 Michael Feldmann, Andreas Padalkin, Christian Scheideler, and Shlomi Dolev. Coordinating amoebots via reconfigurable circuits. *Journal of Computational Biology*, 29(4):317–343, 2022. doi:10.1089/CMB.2021.0363.

8 Roland Flury and Roger Wattenhofer. Slotted programming for sensor networks. In Tarek F. Abdelzaher, Thiemo Voigt, and Adam Wolisz, editors, *Proceedings of the 9th International Conference on Information Processing in Sensor Networks, IPSN 2010, April 12-16, 2010, Stockholm, Sweden*, pages 24–34. ACM, 2010. doi:10.1145/1791212.1791216.

9 Sebastian Forster, Martin Grösbacher, and Tijn de Vos. An improved random shift algorithm for spanners and low diameter decompositions. In Quentin Bramas, Vincent Gramoli, and Alessia Milani, editors, *25th International Conference on Principles of Distributed Systems, OPODIS 2021, December 13-15, 2021, Strasbourg, France*, volume 217 of *LIPIcs*, pages 16:1–16:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.OPODIS.2021.16.

10 Lauri Hella, Matti Järvisalo, Antti Kuusisto, Juhana Laurinharju, Tuomo Lempiäinen, Kerkko Luosto, Jukka Suomela, and Jonni Virtema. Weak models of distributed computing, with connections to modal logic. *Distributed Comput.*, 28(1):31–53, 2015. doi:10.1007/S00446-013-0202-3.

11 Gary L. Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved parallel algorithms for spanners and hopsets. In Guy E. Blelloch and Kunal Agrawal, editors, *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pages 192–201. ACM, 2015. doi:10.1145/2755573.2755574.

12 Gary L. Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In Guy E. Blelloch and Berthold Vöcking, editors, *25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '13, Montreal, QC, Canada - July 23 - 25, 2013*, pages 196–203. ACM, 2013. doi:10.1145/2486159.2486180.

13 Andreas Padalkin and Christian Scheideler. Polylogarithmic time algorithms for shortest path forests in programmable matter. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2024. To appear.

14 Andreas Padalkin, Christian Scheideler, and Daniel Warner. The structural power of reconfigurable circuits in the amoebot model. In Thomas E. Ouldridge and Shelley F. J. Wickham, editors, *28th International Conference on DNA Computing and Molecular Programming, DNA 28, August 8-12, 2022, University of New Mexico, Albuquerque, New Mexico, USA*, volume 238 of *LIPIcs*, pages 8:1–8:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.DNA.28.8.

15 David Peleg. *Distributed computing: a locality-sensitive approach.* SIAM, 2000.

16 David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM J. Comput.*, 30(5):1427–1442, 2000. doi:10.1137/S0097539700369740.

17 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012. doi:10.1137/11085178X.

Lock-Free Augmented Trees

Panagiota Fatourou
FORTH ICS, Heraklion, Greece
University of Crete, Heraklion, Greece

Eric Ruppert
York University, Toronto, Canada

Abstract

Augmenting an existing sequential data structure with extra information to support greater functionality is a widely used technique. For example, search trees are augmented to build sequential data structures like order-statistic trees, interval trees, tango trees, link/cut trees and many others. We study how to design *concurrent* augmented tree data structures. We present a new, general technique that can augment a lock-free tree to add any new fields to each tree node, provided the new fields' values can be computed from information in the node and its children. This enables the design of lock-free, linearizable analogues of a wide variety of classical augmented data structures.

As a first example, we give a wait-free trie that stores a set S of elements drawn from $\{0, \ldots, N-1\}$ and supports linearizable order-statistic queries such as finding the kth smallest element of S. Updates and queries take $O(\log N)$ steps. We also apply our technique to a lock-free binary search tree (BST), where changes to the structure of the tree make the linearization argument more challenging. Our augmented BST supports order statistic queries in $O(h)$ steps on a tree of height h. The augmentation does not affect the asymptotic step complexity of the updates. As an added bonus, our technique supports arbitrary multi-point queries (such as range queries) with the same step complexity as they would have in the corresponding sequential data structure. For both our trie and BST, we give an alternative augmentation to improve searches and order-statistic queries to run in $O(\log |S|)$ steps (at the cost of increasing step complexity of updates by a factor of $O(\log |S|)$).

2012 ACM Subject Classification Theory of computation → Concurrent algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases shared-memory, data structure, tree, binary search tree, augmentation, linearizable, lock-free, order statistic, snapshot

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.23

Related Version *Full Version*: `https://arxiv.org/abs/2405.10506` [24]

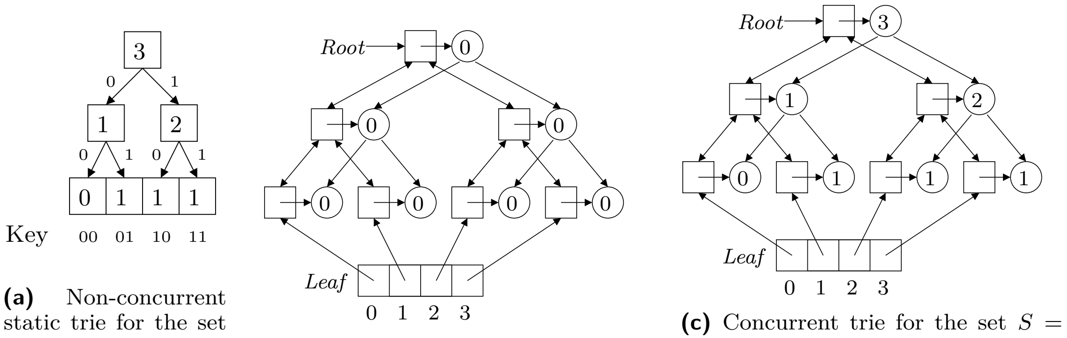
Funding This research was supported by the Hellenic Foundation for Research and Innovation (HFRI) under the "Second Call for HFRI Research Projects to support Faculty Members and Researchers" (project name: PERSIST, project number: 3684) and the Natural Sciences and Engineering Research Council of Canada.

Acknowledgements We thank the anonymous reviewers for their comments.

1 Introduction

Augmentation is a fundamental technique to add functionality to sequential data structures and to make them more efficient, particularly for queries. Augmentation is sufficiently important to warrant a chapter in the algorithms textbook of Cormen et al. [18], which illustrates the technique with the most well-known example of augmenting a binary search tree (BST) so that each node stores the size of the subtree rooted at it. This adds support for many order-statistic queries, including finding the j-th smallest element in the BST or the rank of a given element, *in sub-linear time.* In a balanced BST, these queries take logarithmic time whereas a traversal of an unaugmented BST would take linear time to answer them.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 23; pp. 23:1–23:24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a) Non-concurrent static trie for the set $S = \{1, 2, 3\}$.

(b) Initialization of concurrent trie.

(c) Concurrent trie for the set $S = \{1, 2, 3\}$.

Figure 1 Examples of the trie data structure when $U = \{0, 1, 2, 3\}$, where each node is augmented with a field that stores the number of elements in the subtree. Nodes are shown as squares, Versions are shown as circles containing their *sum* fields.

More generally, each node of a (sequential) tree data structure can be augmented with any number of additional fields that are useful for various applications, provided that the new fields of a node can be computed using information in that node and its children. When applied to many standard trees, such as balanced or unbalanced BSTs, tries or B-trees, the augmentation does not affect the asymptotic time for simple updates, like insertions or deletions, but it can facilitate many other efficient operations. For example, a balanced BST can be augmented for a RangeSum query that computes the sum of all keys within a given range in logarithmic time by adding a field to each node that stores the sum of keys in the node's subtree. (The sum can be replaced by any associative aggregation operator, such as minimum, maximum or product.) Similarly, a BST of key-value pairs can be augmented to aggregate the *values* associated with keys in a given range: each node should store the sum of values in its subtree. One can also *filter* values, for example to obtain the aggregate of all odd values within a range. More sophisticated augmentations can also be used. For instance, an interval tree stores a set of intervals in a balanced BST sorted by the left endpoints, where each node is augmented to store the maximum right endpoint of any interval in the node's subtree, so that one can determine whether any interval in the BST includes a given point in logarithmic time [18]. There are many other types of augmented trees, including one representing piecewise constant functions [12; 13, Section 4.5], measure trees [26], priority search trees [33] and segment trees [9, 10]. Section 3.5 gives another novel example of how to use tree augmentation. Augmented trees are also used as a building block for many other sequential data structures such as link/cut trees [41] and tango trees [19]. These structures have many applications in graph algorithms, computational geometry and databases.

We consider how to augment *concurrent* tree data structures. The resulting data structures are linearizable and lock-free and use single-word compare-and-swap (CAS) instructions. The technique we introduce is very general: as in the sequential setting, it can handle any augmentation to a lock-free tree data structure where the new fields can be computed using the data stored in the node and its children. Thus, it can be used to provide efficient, lock-free shared implementations of many of the sequential data structures mentioned above. Our augmentation does not affect the asymptotic step complexity (i.e., number of steps taken) of update operations. Moreover, we provide a way for queries to obtain a snapshot of the data structure so that they can simply execute the sequential code to answer the query.

For ease of presentation, in Section 3, we first illustrate the technique applied to a simple data structure that represents a dynamically-changing set S of keys drawn from the universe $U = \{0, \ldots, N-1\}$. The basic data structure is a *static* binary trie of height $\log_2 N$, where each key of U is assigned a leaf. To add support for order-statistic queries, each node stores the number of elements of S in its subtree. See Figure 1a for an example. Our technique mirrors this tree of nodes by a tree of Version objects, which store the mutable fields of the augmentation (see Figure 1c). Insertions and deletions of elements modify the appropriate leaf of the tree (and its Version), and then cooperatively propagate any changes to the Version objects stored in ancestors of that leaf until reaching the root. This cooperative approach ensures updates perform a constant number of steps at each node along this path, taking $O(\log N)$ steps in total. Our algorithm never changes fields of Version objects, including their child pointers. Thus, reading the root node's Version object provides a "snapshot" of the entire Version tree, which a query can then explore at its leisure, knowing that it will not be changed by any concurrent updates. Thus, any query operation that follows pointers from the root can be performed exactly as in a sequential version of the data structure, using the same number of steps. For example, order-statistic queries can be answered using $O(\log N)$ steps, and the size of S can be found in $O(1)$ steps. All operations are wait-free.

In Section 4, we describe how to apply the technique to a BST. This has additional complications because the structure of the tree changes as keys are inserted or deleted. We augment the lock-free BST of Ellen et al. [21], which has amortized step complexity $O(h+c)$ per operation, where h is the height of the tree and c is the point contention (i.e., the maximum number of updates running at any point in time). Our augmentation does not affect this asymptotic step complexity of the lock-free update operations, and wait-free queries can again be performed using the same number of steps as in a sequential implementation.

In an augmented tree, each insertion or deletion must typically modify many tree nodes. For example, an insertion in an order-statistic tree must increment the size field in all ancestors of the inserted node. In the concurrent setting, we must ensure that all of these changes appear to take place atomically, so that queries operate correctly. It is generally very difficult to design lock-free data structures where many modifications must appear atomic. Our proposed technique addresses this challenge in a rather simple way. However, the full proofs of correctness are fairly challenging.

Our approach yields a query to find the number of keys in a lock-free tree in $O(1)$ steps. A previous, more general method for adding a size query to any dynamic set [38] is substantially more complicated, and their size queries take $\Omega(P)$ steps in a system of P processes.

Whether augmentation of the tree is needed or not, our technique also provides a simple way of taking a snapshot of the tree to answer queries that must examine multiple locations in the tree, such as a range query. Thus, in addition to supporting augmentation, our technique provides an alternative to other recent work on providing linearizable range queries on concurrent trees [4, 7, 14, 17, 23] or more general snapshots [36, 37, 44]. Ordinarily, our snapshots can be discarded when the query completes, but they can also be used to maintain past versions of the data structure. Many of these other approaches use multiversioning and require complex schemes for unlinking old, obsolete versions from the data structure to facilitate garbage collection (e.g., [8, 45]). The simplicity of our approach avoids this.

2 Related Work

There is very little previous work on concurrent augmented trees. This year, Kokorin et al. [32] described a wait-free BST supporting order-statistic queries and range queries. They use a FIFO queue for each tree node. Before reading or writing a node, an operation must join the node's queue and help each operation ahead of it in the queue by performing that operation's access to the node and, if necessary, adding the operation to the queue of the node's child (or children). This adds $\Omega(Ph)$ to the worst-case step complexity per operation when there are P processes accessing a tree of height h. To handle order-statistic queries, each node stores the size of its subtree. While queueing at the root, an update operation must determine whether it will ultimately succeed (by searching down to a leaf, and checking the queue of pending updates at each node along the way), so that it knows whether to modify the *size* field of nodes as it traverses them. This top-down approach does not seem to generalize to other augmentations where new fields are generally computed bottom-up because the values of the fields of a node usually depend on the values in the node's children.

Independently of this work, Sela and Petrank [39] recently gave a lock-based implementation of an augmented BST. Their approach is restricted to augmentations that compute aggregate functions based on an Abelian group operator (such as sum or product, but not max or min), whereas ours handles arbitrary augmentations. Their approach requires substantial coordination between concurrent operations. Updates are announced, and each query must then take into account information from all ongoing updates with timestamps earlier than its own, using a multiversioning system similar to [23, 44] that maintains version lists at each tree node. In both variants of their algorithm, queries and updates each take at least $\Omega(Ph)$ steps in the worst case when P processes access a tree of height h. Moreover, an update must hold a lock on the nodes where it is performing an insert or delete while it performs $\Omega(Ph)$ steps to update aggregated values.

Sun, Ferizovic and Blelloch [42] discuss augmented trees in a parallel setting, but their focus is on processes sharing the work of a single expensive operation (like a large range query or unioning two trees), whereas our goal is to support multiple concurrent operations.

Independently of this work, Ko [31] used a binary trie structure to add support for predecessor queries to a lock-free data structure for a set drawn from the universe $U = \{0, 1, \ldots, N-1\}$. However, his trie design is quite different from the one we give in Section 3. It supports searches in $O(1)$ steps, while the amortized step complexity for updates and predecessor queries is $O(c^2 + \log N)$, where c is a measure of contention. Thus, searches are faster, but other operations are slower than in our trie. Moreover, Ko's approach does not appear to generalize to other order-statistic queries or other types of augmentations.

The cooperative approach we use to propagate operations up to the root of the tree originates in the universal construction of Afek, Dauber and Touitou [1]. It has been used to build a variety of lock-free data structures [5, 22, 29, 34]. All of these applied the technique to a tournament tree with one leaf per process. A process adds an operation at its leaf, and processes move up the tree gathering larger batches of operations until the batch is applied to the data structure at the root of the tournament tree. Here, we instead apply the approach directly to the tree data structure itself to build larger and larger pieces of the updated tree until we reach the root, at which time we have constructed a new version of the data structure (without destroying any previous versions).

Jayanti [28] used the technique of [1] to implement an array $A[1..n]$ where processes can update an array element and query the value of some fixed function $f(A[1], \ldots, A[n])$, if f can be represented as an evaluation tree similar to a circuit (where leaves are elements of the array, each internal node represents some function of its children and the root represents f). Updates cooperatively propagate changes up the tree so that a query can read f's value from the root. Our trie has some similarities, but is much more general: instead of simply computing a function value, we construct a copy of the data structure that can be used for more complex queries. Our BST implementation goes further to remove the restriction that the shape of the tree being used for the propagation is fixed.

Another technique that cooperatively builds trees bottom-up appears in Chandra, Jayanti and Tan's construction [16] of closed objects (where the effect of any pair of operations is equivalent to another operation). They build trees that represent batches of operations to keep track of the sequence of all operations applied to the closed object being implemented. In contrast, we directly build a representation of the implemented tree data structure.

Our work is on augmenting tree data structures with additional fields to support additional functionality. The main challenge is to make changes to several nodes required by an insert or delete appear atomic. As a byproduct, our technique for doing this also allows processes to take a snapshot of the tree, which can be used to answer arbitrary queries on the state of the tree. For example, it can be used on a BST to find all keys in a given range. A number of recent papers [11, 23, 36, 37, 44] use some form of multiversioning to add the ability to take a snapshot of the state of a concurrent data structure (but without addressing the problem of augmentation). Our approach applies only to trees, whereas some of the other work can be applied to arbitrary data structures, but we do get more efficient queries: a query in our scheme has the same step complexity as the corresponding query in a sequential implementation, whereas a query that runs on top of other multi-versioning schemes, such as that of [44], can take additional steps for every update to the tree that is concurrent with the query. Our approach is more akin to that of functional updates to the data structure that leave old versions accessible, as in the work on classical persistent data structures [20], but the novelty here is that the new versions are built cooperatively by many concurrent operations.

3 Augmented Static Trie

In this section, we illustrate our augmentation technique for a simple data structure that represents a set S of keys drawn from the universe $U = \{0, 1, \ldots, N-1\}$. For simplicity, assume N is a power of 2. A simple, classical data structure for S is a bit vector $B[0..N-1]$, where $B[i] = 1$ if and only if $i \in S$. Even in a concurrent setting, update operations (insertions and deletions of keys) can be accomplished by a single CAS instruction and searches for a key by a single read instruction.

Now, suppose we wish to support the following order-statistic queries.

- Select(k) returns the kth smallest element in S.
- Rank(x) returns the number of elements in S smaller than or equal to x.
- Predecessor(x) returns the largest element in S that is smaller than x.
- Successor(x) returns the smallest element in S that is larger than x.
- Minimum and Maximum return the smallest and largest element in S, respectively.
- RangeCount(x_1, x_2) returns the number of elements in S between x_1 and x_2.
- Size returns $|S|$, the number of elements in the set S.

In the *non-concurrent* setting we can build a binary tree of height $\log_2 N$ whose leaves correspond to the elements of the bit vector, as shown in Figure 1a. We augment each node x with a *sum* field to store the sum of the bits in x's descendant leaves, i.e., the number of elements of S in the subtree rooted at x. For a leaf, the *sum* field is simply the bit that indicates if that leaf's key is present in S. The *sum* field of an internal node can be computed as the sum of its children's *sum* fields. It is straightforward to see that Size queries can then be answered in $O(1)$ steps and the other order-statistic queries can be answered in $O(\log N)$ steps. We call this data structure a *static trie* because the path to the leaf for $i \in U$ is dictated by the bits of the binary representation of i, as in a binary trie [25; 30, Section 6.3]: starting from the root, go left when the next bit is 0, or right when the next bit is 1. Although the trie's *shape* is static, it represents a dynamically changing set S.

3.1 Wait-Free Implementation

The challenge of making the augmented trie concurrent is that each insertion or deletion, after setting the bit in the appropriate leaf, must update the *sum* fields of all ancestors of that leaf. All of these updates must appear to take place atomically. To achieve this, we use a modular design that separates the structure of the tree (which is immutable) from the mutable *sum* fields of the nodes. This modularity means the same approach can be used to augment various kinds of lock-free trees.

We use Node objects to represent the tree structure. Each Node has a *version* field, which stores a pointer to a Version object that contains the current value of the Node's *sum* field. A Version object v associated with a node x also stores pointers *left* and *right* to the Version objects that were associated with x's children at the time when v was created. This way, the Version objects form a *Version tree* whose shape mirrors the tree of Nodes. See Figure 1b. Query operations are carried out entirely within this Version tree. To simplify queries, fields of Versions are immutable, so that when a query reads the root Node's *version*, it obtains a snapshot of the entire Version tree that it can later explore by following child pointers.

To see how updates work, consider an Insert(3) operation, starting from the initial state of the trie shown in Figure 1b. It must increment the *sum* field of the leaf for key 3 and of each Node along the path from that leaf to the root. Since Versions' fields are immutable, whenever we wish to change the data in the Version associated with a Node x, we create a *new* Version initialized with the new *sum* value for the Node, together with the pointers to the two Versions of x's children from which x's *sum* field was computed. Then, we use a CAS to attempt to swing the pointer in $x.version$ to the new Version. If the Insert(3) runs by itself, it would make the sequence of changes shown in Figure 4 as it works its way up the tree. The Insert is linearized when the root Node's *version* field is changed (Figure 4c). Prior to that linearization point, any query operation reading the root's *version* field gets a pointer to the root of the initial Version tree; after it, a query operation gets a pointer to a Version tree that reflects all the changes required by the Insert. A Delete(k) operation is handled similarly by decrementing the *sum* field at each Node along the path from k's leaf Node to the root.

Now, consider concurrent updates. Each update operation must ensure that the root's *version* pointer is updated to reflect the effect of the update. We avoid the performance bottleneck that this could create by having update operations *cooperatively* update Versions. At each Node x along the leaf-to-root path, the update reads the *version* field from both of x's children, creates a new Version for x based on the information in the children's Versions, and attempts to install a pointer to it in $x.version$ using a CAS. Following the terminology of [28], we call this procedure a *refresh*. This approach is cooperative, since a refresh of Node x by one update will propagate information from *all* updates that have reached either child of x to x. If an update's first refresh on x fails, it performs a second refresh. This is called a *double refresh* of x. We shall show that attempting a refresh twice at each Node suffices: if both of the CAS steps in an update's *double refresh* on a Node x fail, it is guaranteed that some other process has propagated the update's information to x.

Figure 2 describes the fields of our objects. Figure 3 provides pseudocode for the implementation. It is substantially simpler than previous lock-free tree data structures for sets, even though it includes augmentation and provides atomic snapshots. In our code, if *ptr* is a pointer to an object O, $ptr.f$ denotes field f of O. A shared pointer *Root* points to the root Node of the binary tree with N leaves. To expedite access to the leaves, we use an array $Leaf[0..N-1]$, where $Leaf[k]$ points to the leaf Node for key k.

```
type Node                          ▷ used to store nodes of static trie structure
    Node* left, right              ▷ immutable pointers to children Nodes
    Node* parent                   ▷ immutable pointer to parent Node
    Version* version               ▷ mutable pointer to current Version

type Version                       ▷ used to store a Node's augmented data
    Version* left, right           ▷ immutable pointers to children Versions
    int sum                        ▷ immutable sum of descendant leaves' bits
```

Figure 2 Object types used in wait-free trie data structure.

A **Refresh**(x) reads the *version* field of x and its two children, creates a new Version for x based on information in the children's Versions, and then attempts to CAS the new Version into $x.version$. To handle different augmentations, one must only change the way Refresh computes the new fields. **Propagate**(x) performs a double Refresh at each node along the path from x to the root.

An **Insert**(k) first checks if the key k is already present in the set at line 15. If not, it uses a CAS at line 18 to change the leaf's Version object to a new Version object with *sum* field equal to 1. If the CAS succeeds, the Insert will return true to indicate a successful insertion. If the key k is already present when the read at line 14 is performed or if the CAS fails (meaning that some concurrent operation has already inserted k), the Insert will return false. In all cases, the Insert calls Propagate before returning to ensure that the information in the leaf's Version is propagated all the way to the root.

The **Delete**(k) operation is very similar to an insertion, except that the operation attempts to switch the *sum* field of $Leaf[k]$ from 1 to 0.

Find and **Select** are given as examples of query operations. They first take a snapshot of the Version tree by reading $Root.version$ on line 41 or 47 and then execute the query's standard sequential code on that tree. Other queries can be done similarly. In particular, to ensure linearizability, queries should access the tree only via $Root.version$, not through the *Leaf* array.

3.2 Correctness

A detailed proof of correctness appears in [24]; we sketch it here. We first look at the structure of Version trees. Let U_x be the sequence of keys from the universe U that are represented in the subtree rooted at Node x of the tree, in the order they appear from left to right. In particular, $U_{Root} = \langle 0, 1, \ldots, N-1 \rangle$. It can be shown by induction on the height of the Node x, that the Version tree rooted at $x.version$ is a perfect binary tree with $|U_x|$ leaves. Recall that the fields of Version objects are immutable, so the proof must only consider lines 17, 25 and 33, which create new Version objects. The induction step can be easily proved because of the way the Version tree for x is constructed at line 33 by combining the Version trees for x's children. Line 33 also ensures that we maintain as an invariant that,

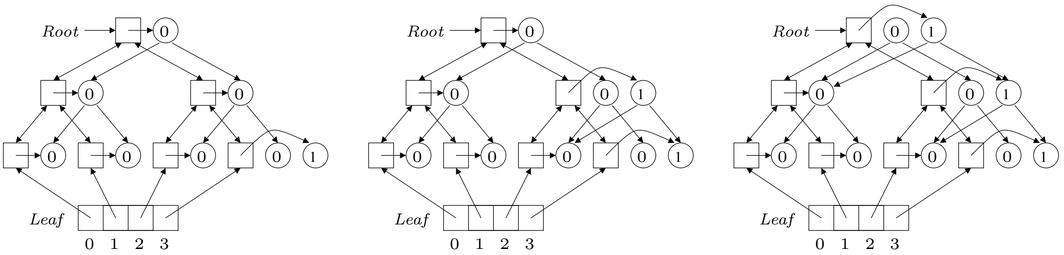
$$\text{for every internal Version } v, v.sum = v.left.sum + v.right.sum. \tag{1}$$

Since leaf Versions contain 0 or 1 (according to lines 17 and 25), $v.sum$ stores the sum of the bits stored in leaves of the subtree rooted at v.

The key goal of the correctness proof is to define linearization points for the update operations (insertions and deletions) so that, at all times, the Version tree rooted at $Root.version$ accurately reflects all update operations linearized so far. Then, we linearize each query

```
Initialization (refer to Figure 1b):
Node* $Root$ ← root of a perfect binary tree of Nodes with $N$ leaves.
For each Node $x$, $x.version$ points to a new Version with fields $sum$ ← 0, $left$ ← $x.left.version$
                  and $right$ ← $x.right.version$.
Node* $Leaf[0..N-1]$ contains pointers to the leaf Nodes of the binary tree.

Insert(int $k$) : Boolean                ▷ Add $k$ to $S$; return true iff $k$ was not already in $S$
    $old$ ← $Leaf[k].version$
    $result$ ← $(old.sum = 0)$
    if $result$ then
        $new$ ← new Version with $sum$ ← 1, $left$ ← Nil, and $right$ ← Nil
        $result$ ← CAS($Leaf[k].version, old, new$)
    Propagate($Leaf[k].parent$)
    return $result$

Delete(int $k$) : Boolean                ▷ Remove $k$ from $S$; return true iff $k$ was in $S$
    $old$ ← $Leaf[k].version$
    $result$ ← $(old.sum = 1)$
    if $result$ then
        $new$ ← new Version with $sum$ ← 0, $left$ ← Nil and $right$ ← Nil
        $result$ ← CAS($Leaf[k].version, old, new$)
    Propagate($Leaf[k].parent$)
    return $result$

Refresh(Node* $x$) : Boolean             ▷ Try to propagate information to $x$ from its children
    $old$ ← $x.version$
    $v_L$ ← $x.left.version$
    $v_R$ ← $x.right.version$
    $new$ ← new Version with $left$ ← $v_L$, $right$ ← $v_R$, $sum$ ← $v_L.sum + v_R.sum$
    return CAS($x.version, old, new$)

Propagate(Node* $x$)                     ▷ Propagate updates from $x$'s children up to root
    while $x$ is not Nil do
        if not Refresh($x$) then
            Refresh($x$)                 ▷ Do a second Refresh if first one fails
        $x$ ← $x.parent$

Find(Key $k$) : Boolean                  ▷ Check if key $k$ is in $S$
    $v$ ← $Root.version$                 ▷ Start at the root
    for $i$ ← $1..\log_2 N$ do           ▷ Traverse path to leaf of Version tree
        if $i$th bit of binary representation of $k$ is 0 then $v$ ← $v.left$
        else $v$ ← $v.right$
    return $(v.sum = 1)$

Select($j$) : int                        ▷ Return the $j$th smallest element in $S$
    $v$ ← $Root.version$                 ▷ Start at the root
    $i$ ← 1                              ▷ Keep track of breadth-first index of $v$ in tree
    if $v.sum < j$ then return Nil       ▷ No such element in $S$
    else
        while $v.left \neq$ Nil do
            if $v.left.sum \geq j$ then  ▷ Required element is in left subtree
                $v$ ← $v.left$
                $i$ ← $2i$
            else                         ▷ Required element is in right subtree
                $v$ ← $v.right$
                $i$ ← $2i + 1$
                $j$ ← $j - v.left.sum$   ▷ Adjust rank of element being searched for
        return $i - N$                   ▷ Convert breadth-first index to value
```

Figure 3 Implementation of wait-free augmented trie.



(a) After the CAS on line 18 updates the *version* of the rightmost leaf Node.

(b) After the CAS on line 34 updates the right child of the root Node in the first iteration of the loop in Propagate.

(c) After the CAS on line 34 updates the root Node in the second iteration of the loop in Propagate.

Figure 4 Key steps of an Insert(3) into the initially empty set shown in Figure 1b.

operation at the time it reads *Root.version* to take a snapshot of the Version tree. This will ensure that the result returned by the query is consistent with the state of the represented set S at the query's linearization point.

We consider an execution in which processes perform operations on the trie. An execution is formalized as an alternating sequence of configurations and steps $C_0, s_1, C_1, s_2, \ldots$, where each configuration C_i describes the state of the shared memory and the local state of each process, and each s_i is a step by some process that takes the system from configuration C_{i-1} to C_i. A step is either a shared-memory access or a local step that affects only the process's local state. C_0 is the initial configuration described in lines 8–12.

Our goal is to define a linearization point (at a step of the execution) of each update operation so that for each configuration C, the Version tree rooted at *Root.version* is the trie that would result by sequentially performing all the operations that are linearized before C in their linearization order. Thus, the linearization point of an update operation should be the moment when the effect of the update has been propagated to the root Node, so that it becomes visible to queries. To define these linearization points precisely, we define the *arrival point* of an update operation on a key k at each Node along the path from the leaf Node representing k up to the root Node. Intuitively, the arrival point of the update at Node x is the moment when the effect of the update is reflected in the Version tree rooted at $x.version$. Then, the linearization point is simply the arrival point of the update at *Root*. We must ensure these linearization points are well defined by showing that the double-refresh technique propagates each update all the way up to *Root* before the update terminates.

Definition 1, below, formally defines the arrival point of each Insert(k) or Delete(k) operation at Node x, where $k \in U_x$ using induction from the bottom of the tree to the top. If an Insert(k) sees that k is already in a leaf Node at line 14, or if a Delete(k) sees that k is not present in a leaf Node at line 22, the arrival point of the operation is at that line. Otherwise the update performs a CAS on the leaf at line 18 or 26. If the CAS succeeds, the CAS is the update's arrival point at that leaf. Otherwise, we put the arrival point of the update at the leaf at a time when k's presence or absence would cause the update to fail. An update's arrival point at an internal Node is the first successful CAS by a Refresh that previously read the child after the update's arrival point at that child.

▶ **Definition 1.** *We first define the arrival point of an Insert(k) or Delete(k) operation op at Leaf[k].*

1. *If op performs a successful CAS at line 18 or 26, then the arrival point of op is that CAS.*
2. *If op performs an unsuccessful CAS at line 18 or 26, then the arrival point of op is the first successful CAS on Leaf[k].version after op read the old value of Leaf[k].version at line 14 or 22. (Such a CAS must exist; otherwise op's CAS would have succeeded.)*

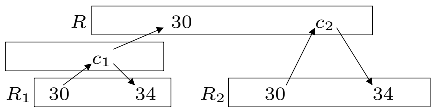


Figure 5 Calls to Refresh in proof that a double refresh successfully propagates updates to a Node from its children. The horizontal axis represents time, and boxes indicate the interval between a routine's invocation and its response. Numbers refer to line numbers in the pseudocode. An arrow $s_1 \rightarrow s_2$ indicates step s_1 must precede step s_2.

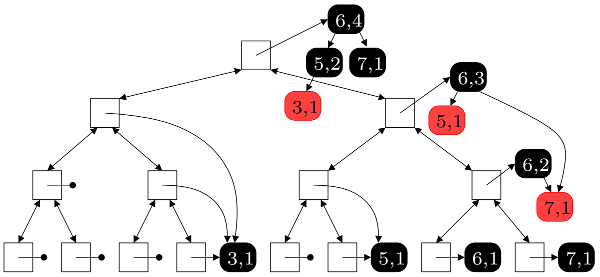


Figure 6 Augmenting the trie with red-black trees (RBTs) to speed up queries. $N = 8$ and $S = \{3, 5, 6, 7\}$. Squares are trie Nodes. Ovals are RBT nodes. Each RBT node has child pointers, and stores a key and a *size* field that represents the number of keys in the subtree. Black dots represent RBT nodes with *sum* 0.

3. *If op is an* Insert *that reads a Version with sum* $= 1$ *from Leaf*$[k]$*.version on line 14 or op is a* Delete *that reads a Version with sum* $= 0$ *from Leaf*$[k]$*.version on line 22, then the arrival point of op is op's read at line 14 or 22, respectively.*

If multiple operations' arrival points at a leaf Node are at the same successful CAS*, we order them: first the operation that did the successful* CAS*, then all the other operations (ordered arbitrarily).*

Next, we define the arrival point of an Insert(k) *or* Delete(k) *op at an internal Node* x *with* $k \in U_x$*.*

4. *If* $k \in U_{x.left}$*, the arrival point of op is the first successful* CAS *on* $x.version$ *at line 34 of a* Refresh *that read* $x.left.version$ *at line 31 after the arrival point of op at* $x.left$*.*
5. *If* $k \in U_{x.right}$*, the arrival point of op is the first successful* CAS *on* $x.version$ *at line 34 of a* Refresh *that read* $x.right.version$ *at line 32 after the arrival point of op at* $x.right$*.*

If multiple operations' arrival points at an internal Node are at the same successful CAS*, we order them as follows: first the operations on keys in* $U_{x.left}$ *in the order they arrived at* $x.left$ *and then the operations on keys in* $U_{x.right}$ *in the order they arrived at* $x.right$*.*

For example, consider the Insert(3) depicted in Figure 4. Its arrival point at the leaf Node for key 3 is the CAS that updates that leaf's *version* field, shown in Figure 4a. Its arrival point at the parent of this leaf is the CAS that updates the data structure as shown in Figure 4b. Its arrival point at the root is the CAS that updates the *Root.version* as shown in Figure 4c.

It follows easily from Definition 1 that arrival points of an update operation *op* are after *op* begins. If *op* terminates, we must also show that it has an arrival point at the root Node before it terminates. Recall that after *op*'s arrival point at a leaf, *op* calls Propagate, which does a double Refresh at each Node along the path from that leaf to the root. We show by induction that the double refresh at each node x along the path ensures *op* has an arrival point at x. The induction step follows immediately from Parts 4 and 5 of Definition 1 if one of *op*'s calls to Refresh(x) performs a successful CAS. So, suppose both of x's calls R_1 and R_2 to Refresh(x) fail their CAS. Then for each R_i, there must be a successful CAS c_i on $x.version$ between R_i's read of $x.version$ on line 30 and its CAS on line 34, as depicted in Figure 5. Although c_1 may store outdated information, the Refresh that performs c_2 must have read information from x's children after c_1, which is enough to ensure that *op* has an arrival point at x, by Parts 4 and 5 of Definition 1.

Our next goal is to prove a key invariant that, for each configuration C and Node x, the Version tree rooted at $x.version$ accurately reflects all of the updates whose arrival points at x are prior to C. In other words, it is a trie structure (similar to the one shown in Figure 1a) that would result from performing all of those updates in the order of their arrival points at x. As a corollary, when we take x to be the root Node, we see that the Version tree rooted at $Root.version$ has a 1 in the leaf for key k if and only if k is in the set obtained by sequentially performing the linearized operations in order. Correctness of all query operations follows from this fact and the invariant (1).

We sketch the proof of the key invariant. We make the argument separately for each key $k \in U_x$. We define $Ops(C, x, k)$ to be the sequence of update operations on key k whose arrival points at x precede configuration C, in the order of their arrival points. We must show that, in each configuration C, the leaf corresponding to key k in the subtree rooted at $x.version$ contains a 1 if and only if $Ops(C, x, k)$ ends with an Insert(k).

If x is the leaf for key k, we consider each step that can add arrival points at x. First, consider a CAS that flips the bit stored in $x.version$. If the CAS sets the bit to 1, it follows from Part 1 and Part 2 of Definition 1 that it is the arrival point of one or more Insert(k) operations, which preserves the invariant. Similarly, a CAS that sets the bit to 0 is the arrival point of one or more Delete(k) operations, which preserves the invariant. If the step is an Insert(k)'s read of $x.version$ when it has value 1 or a Delete(k)'s read of $x.version$ when it has value 0, it also preserves the invariant.

If x is an internal Node, the fact that the invariant holds at x can be proved inductively. The claim at x follows from the assumption that it holds at the children of x, since the invariant is phrased in terms of a single key and the sets of keys represented in the two subtrees of x are disjoint.

Finally, we prove that operations that arrive at a leaf are propagated up the tree in an orderly way, so that they arrive at the root in the same order. This is useful for showing that the update operations return results consistent with their linearization order.

3.3 Complexity and Optimizations

Insert and Delete take $O(\log N)$ steps. Searches and the order-statistic queries listed at the beginning of Section 3 take $O(\log N)$ steps and are read-only. Size queries can be answered in $O(1)$ steps by simply returning $Root.version.sum$. We could also augment the data structure so that each node stores the minimum element in its subtree to answer Minimum queries in $O(1)$ steps. A range query that returns R elements can be done in $O(R(\log \frac{N}{R} + 1))$ steps, since it visits at most R locations in the top $\log R$ levels of the Version tree and in the rest of the tree it visits $O(\log N - \log R)$ locations per returned element, for a total of $O(R(\log N - \log R + 1))$ locations. All operations are wait-free.

We assume a safe garbage collector, such as the one provided by Java, which deallocates objects only when they are no longer reachable. We now give a very pessimistic worst-case bound on the space used by objects that are still reachable. For each Node x, up to $O(\log N)$ different Versions belonging to x could be in the Version trees of each of x's ancestors. Thus, the space used by all objects reachable by following pointers from $Root$ is $O(N \log N)$. In addition, any old ongoing queries could have an old snapshot of a Version tree.

The Node tree is static and complete, so it can be represented using an array $Tree[1..2N-1]$ of pointers to Versions, where $Tree[1]$ is the root, and the children of the internal Node $Tree[i]$ are $Tree[2i]$ and $Tree[2i+1]$ [30, p. 144]. This saves the space needed for the $Leaf$ array and parent and child pointers, since we can navigate the tree by index arithmetic rather than following pointers.

3.4 Variants and Other Applications

Generalizing our implementation to d-ary trees is straightforward for any $d \geq 2$. The number of CAS instructions per update would be reduced to $2\log_d N$, but the number of reads (and local work) per update would increase to $\Theta(d \log_d N)$. Order-statistic queries could run in $\Theta(\log_2 N)$ steps if each node stores prefix sums and uses binary search.

Instead of storing a set of keys $S \subseteq U$, a straightforward variant of our data structure can store a set of key-value pairs, where each record has a unique key drawn from U. Instead of storing just one bit, a leaf's Version object would also store the associated value. A Replace(k, v) operation that replaces the value associated with key k with a new value v would update the appropriate leaf's *version* field and call Propagate. If several Replace$(k, *)$ operations try to update a leaf concurrently, one's CAS will succeed and the others will fail, and we can assign them all arrival points at the leaf at the time of the successful CAS, with the failed operations preceding the successful one.

Our approach can also provide lock-free *multisets* of keys drawn from U. Instead of storing a bit, the leaf for key k stores a Version whose *sum* field is the number of copies of k in the multiset. With CAS instructions, operations can be made lock-free if each Insert(k) or Delete(k) repeatedly tries to install a new Version k's leaf with its *sum* field incremented or decremented and then calls Propagate. If the leaf's *sum* field can be updated with a fetch&add, the updates can be made wait-free.

We described how to augment the trie with a *sum* field to facilitate efficient order-statistic queries. However, the method can be used for any augmentation where the values of a node's additional fields can be computed from information in the node and its children, by modifying line 33 to compute the new fields. Section 1 mentions some of the many applications where this can be applied.

Without any modification, our trie supports multipoint queries, like range searches that return all keys in a given range, since reading *Root.version* yields a snapshot of the trie. In fact, our technique has more efficient queries than some recent papers discussed in Section 2 that provide multipoint queries: in our approach, queries take the same number of steps as in a sequential implementation.

3.5 Improving Query Step Complexity to $O(\log |S|)$

The step complexity of order-statistic queries on the set S can be improved from $O(\log N)$ to $O(\log |S|)$. To do this, we simply use a different augmentation. The *version* field of each Node x stores a pointer to the root of a red-black tree (RBT) that represents all the elements in the subtree of Nodes rooted at x. See Figure 6 for an example. A Refresh(x) updates *x.version* by reading the RBTs stored in *x.left.version* and *x.right.version*, joining them into one RBT (without destroying the smaller RBTs) and then using a CAS to store the root of the joined RBT in *x.version*. The algorithm to Join two RBTs in logarithmic time, provided that all elements in one are smaller than all elements in the other, is in Tarjan's textbook [43]. To avoid destroying the smaller RBTs when performing a Join, one can use the path-copying technique of Driscoll et al. [20]. (Path copying has proved useful for a number of concurrent data structures, e.g., [3, 5, 6, 34].) For complete pseudocode, see [24].

Each RBT node also has a *size* field storing the number of elements in the subtree rooted at that node. A query reads *Root.version* to get a snapshot of a RBT containing all elements in the dynamic set. Order-statistic queries are answered in $O(\log |S|)$ steps using the *size* fields of the RBT.

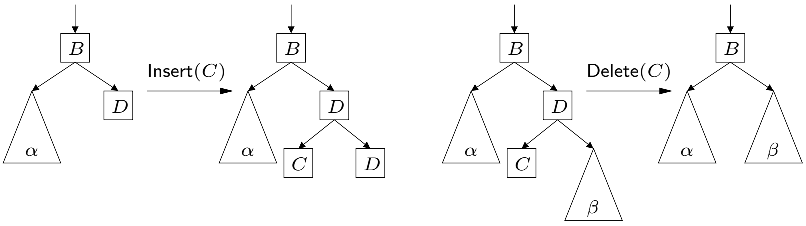


Figure 7 How updates modify a leaf-oriented BST. Here, α and β represent arbitrary subtrees.

There is a tradeoff: the step complexity of updates increases to $O(\log N \log \hat{n})$, where $\hat{n}$ denotes a bound on the maximum size the set S could have under any possible linearization of the update operations. This holds because a Join of two RBTs must be performed at each of $\log N$ Nodes of the Node tree during Propagate. The elements in a RBT constructed by a Refresh on a non-root Node may never all be in the set simultaneously, so we must argue that the size of each such RBT is $O(\hat{n})$. Consider a Join(T_1, T_2) during a call R to Refresh(x). Without loss of generality, assume $|T_1| \geq |T_2|$. Let α' be the prefix of the execution up to the time R reads T_1 from $x.left.version$. Suppose we modify α' by delaying R's read of $x.version$ until just before R reads $x.left.version$, and then appending to the execution all the steps needed to complete the Propagate that called R. This will ensure that all remaining CAS steps of the Propagate succeed and T_1 will be a subtree of the tree stored in $Root.version$. Thus, there must be some way to linearize α' so that all elements in T_1 are simultaneously in the represented set (since the modified execution is linearizable), so $|T_1| \leq \hat{n}$. Thus, the size of the RBT that R builds is $|T_1| + |T_2| \leq 2|T_1| \leq 2\hat{n}$.

4 Augmented Binary Search Tree

In this section, we illustrate our technique by augmenting a binary search tree (BST) that represents a set S of elements drawn from an *arbitrary* (ordered) universe U. We describe the augmentation for order-statistic queries, but as explained above, the same approach can be used for many other applications. In constrast to the augmented trie of Section 3, the step and space complexity of our augmented BST depend on $|S|$ rather than $|U|$.

4.1 Basic Lock-free BST

We base our augmented BST on the lock-free BST of Ellen et al. [21], so we first give a brief overview of how this BST works. The BST is leaf-oriented: keys of S are stored in the leaves; keys in internal nodes serve only to direct searches to the leaves. The BST property requires that all keys in the left subtree of a node x are smaller than x's key and all keys in the right subtree of x are greater than or equal to x's key. The tree nodes maintain child pointers, but not parent pointers. To simplify updates, the BST is initialized with three sentinel nodes: an internal node and two leaves containing dummy keys ∞_1 and ∞_2, which are considered greater than any actual key in U and are never deleted. A shared $Root$ pointer points to the root node of the tree, which never changes.

An Insert or Delete operation starts at the root and searches for the leaf at which to apply its update. Updates are accomplished by simple modifications to the tree structure as shown in Figure 7. To coordinate concurrent updates to the same part of the tree, updates must flag a node before modifying one of its child pointers and remove the flag when the modification is done. Before removing an internal node from the tree, the operation must *permanently* flag it. Since only one operation can flag a node at a time, flagging a node is analogous to locking it. To ensure lock-free progress, an update that needs to flag a node that is already flagged for another update first *helps* the other update to complete and then tries again to perform its operation. When retrying, the update does not begin all over from the top of the tree; the update keeps track of the sequence of nodes it visited on a thread-local stack so that it can backtrack a few steps up the tree by popping the stack until reaching a node that is not permanently flagged for deletion, and then searches onward from there for the location to retry its update. Each update is linearized at the moment one of the changes shown in Figure 7 is made to the tree, either by the operation itself or by a helper.

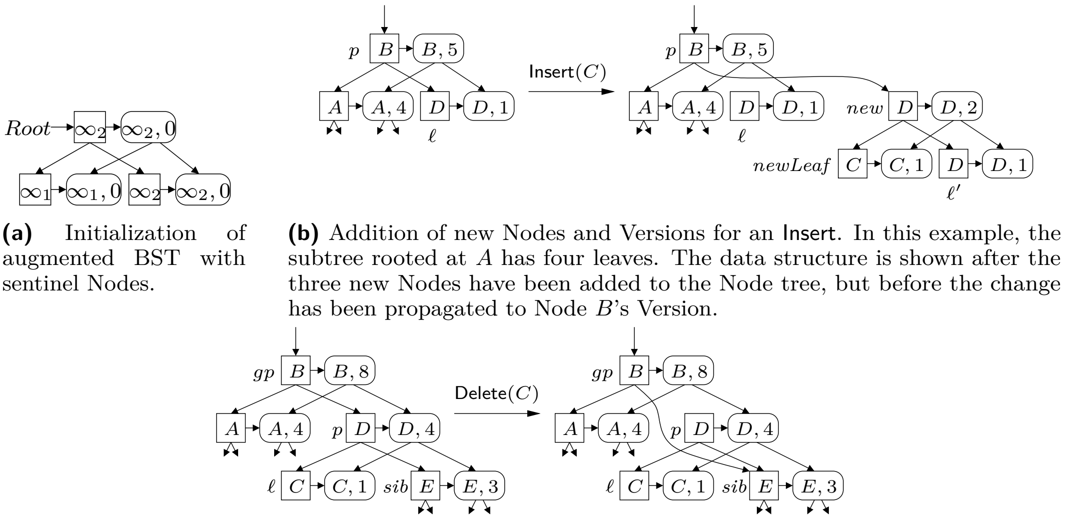
The tree satisfies the BST property at all times. We define the *search path* for a key k at some configuration C to be the path that a sequential search for k would take if it were executed without interruption in C. Searches in the lock-free BST ignore flags and simply follow child pointers until reaching a leaf. A search for k may pass through nodes that get removed by concurrent updates, but it was proved in [21] that each Node the search visits *was* on the search path for k (and by the way we linearize updates, it was thus also in the set represented by the BST) at some time during the search. A search that reaches a leaf ℓ is linearized when that leaf was on the search path for k.

4.2 Lock-free Augmentation

We now describe how to augment the lock-free BST of [21] with additional fields for each node, provided the fields can be computed from information in the node and its children. We again use the *sum* field, which supports efficient order-statistic queries, as an illustrative example. As in Section 3.1, we add to each tree Node x a new *version* field that stores a pointer to a tree of Version objects. This Version tree's leaves form a snapshot of the portion of S stored in the subtree rooted at x. In particular, the leaves of the Version tree stored in *Root.version* form a snapshot of the entire set S. Each Version v stores a *sum* field and pointers to the Versions of x's children that were used to compute v's *sum*. Each Version associated with Node x also stores a copy of x's key to direct searches through the Version trees. Version trees will always satisfy the BST property, and the *sum* field of each Version v stores the number of keys in leaf descendants of v. See Figure 9 on page 20 for a formal description of the Node and Version object types. See Figure 8a for the initial state of the BST, including the sentinel Nodes. Pseudocode for the implementation is in Appendix A.

An **Insert** or **Delete** first runs the algorithm from [21] to modify the Node tree as shown in Figure 7. Figures 8b and 8c show the effects of the modification when Versions are also present. Then, the update calls Propagate to modify the *sum* fields of the Versions of all Nodes along the path from the location where the key was inserted or deleted to the root. As in Section 3.1, an update operation's changes to the *sum* field of all these Nodes become visible at the same time, and we linearize the update at that time. If an Insert(k) reaches a leaf Node that already contains k, before returning false, it also calls Propagate to ensure that the operation that inserted the other copy of key k has been propagated to the *Root* (and therefore linearized). Similarly, a Delete(k) that reaches a leaf Node and finds that k is absent from S also calls Propagate before returning false.

The **Propagate** routine is similar to the one in Section 3.1. As mentioned in Section 4.1, each update uses a thread-local stack to store the Nodes that it visits on the way from *Root* to the location where the update must be performed, so Propagate can simply pop these Nodes off the stack and perform a double Refresh on each of them. Some of these Nodes may have been removed from the Node tree by other Delete operations that are concurrent with the update, but there is no harm in applying a double Refresh to those deleted Nodes.



(a) Initialization of augmented BST with sentinel Nodes.

(b) Addition of new Nodes and Versions for an Insert. In this example, the subtree rooted at A has four leaves. The data structure is shown after the three new Nodes have been added to the Node tree, but before the change has been propagated to Node B's Version.

(c) Change to the Node tree for a Delete. The subtrees rooted at A and E have four and three leaves, respectively. The data structure is shown after the Nodes C and D have been removed from the Node tree, but before the change has been propagated to Node B's Version.

Figure 8 Augmented BST data structure. Nodes are shown as squares and Version objects as ovals with *key* and *sum* fields shown.

As in Section 3.1, each **Refresh** on a Node x reads the Versions of x's children and combines the information in them to create a new Version for x, and then attempts to CAS a pointer to that new Version into $x.version$. There is one difference in the Refresh routine: because x's child pointers may be changed by concurrent updates, Refresh reads x's child pointer, reads that child's *version* field, and then reads x's child pointer again. If the child pointer has changed, Refresh does the reads again, until it gets a consistent view of the child pointer and the *version* field of that child. (It may be that this re-reading could be avoided, but it simplifies the proof of correctness.)

A **query operation** first reads *Root.version* to get the root of a Version tree. This Version tree is an immutable BST (with *sum* fields) whose leaves form a snapshot of the keys in S at the time *Root.version* is read. The query is linearized at this read. The standard, sequential algorithm for an order-statistic query can be run on that Version tree. To ensure linearizability, searches are performed like other queries. This also makes searches wait-free, unlike the original BST of [21], where searches can starve. Complex queries, like range queries, can access any subset of Nodes in the snapshot. Our technique provides snapshots in a simpler way than [23] (later generalized by [44] to any CAS-based data structure), which keeps a list of previous timestamped versions of each child pointer. Our approach makes queries more efficient since they do not have to search back through version lists for an old version with a particular timestamp. It also avoids many of the problems of garbage collection, since old Versions are automatically disconnected from our data structure when a new Version replaces it. Unlike [23], our approach does not provide a snapshot of the Node tree: the shape of the Version tree may not match the shape of the Node tree at any time. Instead, our approach provides a snapshot of the *set of elements represented by the tree.*

Pseudocode for the augmented BST appears in Appendix A and a sketch of the correctness proof is in Appendix B. For a detailed correctness proof, see [24].

4.3 Complexity

The amortized step complexity per operation on the unaugmented BST is $O(h+c)$, where h is the height of the Node tree and c is point contention [21]. Since we have not made any change to the way the Node tree is handled, we must just count the additional steps required for the augmentation. We argue that the amortized step complexity to perform a Propagate is also $O(h+c)$. The number of iterations of the loop in Propagate is bounded by the number of elements pushed on to the stack by the update, which in turn is bounded by the step complexity of the update in the original algorithm of [21]. Recall that a Refresh may have to reread child pointers repeatedly until it gets a consistent view of the child pointer and the child's *version* field. Rereading is necessary only if the child pointer changes between two successive reads. Thus, there are at most c re-reads caused by each change to a child pointer (namely by those Refresh operations running when the change happens). Moreover, there is at most one child pointer change for each update operation. Thus, the amortized step complexity per update operation remains $O(h+c)$. Since queries begin by taking a snapshot of the Version tree, queries are wait-free and take the same number of steps that they would in the sequential setting. For example, searches and order-statistic queries take $O(h)$ steps.

4.4 Extensions

The variants of the trie described in Section 3.4 apply equally to the BST.

The approach of Section 3.5 can be applied to our BST in exactly the same way so that, even though the Node tree is unbalanced, *Root.version* points to a *balanced* Version tree containing the elements of the set. This facilitates queries that can be done in the same number of steps as in a sequential augmented balanced BST. For example, order-statistic queries can all be answered in $O(\log n)$ steps where n is the size of the set. This does, however, increase the amortized step complexity for update operations, which can be bounded using the argument of Section 3.5 by $O((h+c)\log \hat{n})$, where $\hat{n}$ is a bound on the size of the set under any possible linearization of the execution.

5 Future Work

Our technique can provide lock-free implementations of many tree data structures based on augmented trees supporting insertions, deletions, and arbitrarily complex queries.

Although we base our augmented BST on [21], we believe our technique could also be applied to the similar lock-free BST design of Natarajan, Ramachandran and Mittal [35] or other concurrent trees. It would be interesting to apply it to a node-oriented tree such as [27], a balanced tree such as the lock-free chromatic BST of [15] or to a self-balancing concurrent tree such as the CB Tree [2]. In particular, the latter two would require ensuring the Propagate routine works correctly with rotations used to rebalance the tree. The technique may also be applicable to trees that use other coordination mechanisms, such as locks (e.g., [35]).

Could our technique be extended to obtain lock-free implementations of sequential augmented data structures that require more complex updates (such as the insertion of a pair of keys)? In the sequential setting, examples of such data structures include link/cut trees [41] and segment trees [9, 10]. Shafiei [40] described a mechanism for making multiple changes to a tree appear atomic, but it would require additional work to find a suitable way to generalize our Propagate routine with her approach.

References

1 Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proc. 27th ACM Symposium on Theory of Computing*, pages 538–547, New York, NY, USA, 1995. doi:10.1145/225058.225271.

2 Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert Endre Tarjan. The CB tree: a practical concurrent self-adjusting search tree. *Distributed Computing*, 27(6):393–417, 2014. doi:10.1007/S00446-014-0229-0.

3 Vitaly Aksenov, Trevor Brown, Alexander Fedorov, and Ilya Kokorin. Poster: Unexpected scaling in path copying trees. In *Proc. 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 438–440, 2023. doi:10.1145/3572848.3577512.

4 Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. In *Proc. 23rd ACM Symposium on Principles and Practice of Parallel Programming*, pages 14–27, 2018. doi:10.1145/3178487.3178489.

5 Shalom Asbell and Eric Ruppert. A wait-free deque with polylogarithmic step complexity. In *Proc. 27th International Conference on Principles of Distributed Systems*, volume 286 of *LIPIcs*, pages 17:1–17:22, 2023. doi:10.4230/LIPICS.OPODIS.2023.17.

6 Benyamin Bashari and Philipp Woelfel. An efficient adaptive partial snapshot implementation. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 545–555, 2021. doi:10.1145/3465084.3467939.

7 Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. *ACM Trans. Parallel Comput.*, 7(3):16:1–16:28, June 2020. doi:10.1145/3399718.

8 Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. Space and time bounded multiversion garbage collection. In *Proc. 35th International Symposium on Distributed Computing*, volume 209 of *LIPIcs*, pages 12:1–12:20, 2021. doi:10.4230/LIPICS.DISC.2021.12.

9 J. L. Bentley. Solutions to Klee's rectangle problems. Technical report, Carnegie-Mellon University, Pittsburgh, PA, 1977.

10 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

11 Guy E. Blelloch and Yuanhao Wei. VERLIB: concurrent versioned pointers. In *Proc. 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 200–214, 2024. doi:10.1145/3627535.3638501.

12 Prosenjit Bose, Marc J. van Kreveld, Anil Maheshwari, Pat Morin, and Jason Morrison. Translating a regular grid over a point set. *Computational Geometry*, 25(1–2):21–34, 2003. doi:10.1016/S0925-7721(02)00128-1.

13 Peter Brass. *Advanced Data Structures*. Cambridge University Press, 2008.

14 Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In *Proc. 16th International Conference on Principles of Distributed Systems*, volume 7702 of *LNCS*, pages 31–45, 2012. doi:10.1007/978-3-642-35476-2_3.

15 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proc. 19th ACM Symposium on Principles and Practice of Parallel Programming*, pages 329–342, 2014. doi:10.1145/2555243.2555267.

16 Tushar Deepak Chandra, Prasad Jayanti, and King Tan. A polylog time wait-free construction for closed objects. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*, pages 287–296, 1998. doi:10.1145/277697.277753.

17 Bapi Chatterjee. Lock-free linearizable 1-dimensional range queries. In *Proc. 18th International Conference on Distributed Computing and Networking*, pages 9:1–9:10, 2017. doi:10.1145/3007748.3007771.

18 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 17, pages 480–496. MIT Press, fourth edition, 2022.

19 Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic optimality–almost. *SIAM Journal on Computing*, 37(1):240–251, 2007. doi:10.1137/S0097539705447347.

20 James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989. doi:10.1016/0022-0000(89)90034-2.

21 Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing*, pages 332–340, 2014. Full version available online from https://users.ics.forth.gr/~faturu/BSTproof.pdf. doi:10.1145/2611462.2611486.

22 Panagiota Fatourou and Nikolaos D. Kallimanis. The RedBlue family of universal constructions. *Distributed Computing*, 33(6):485–513, 2020. doi:10.1007/S00446-020-00370-7.

23 Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 275–286, 2019. doi:10.1145/3323165.3323197.

24 Panagiota Fatourou and Eric Ruppert. Lock-free augmented trees. Full version available from https://arxiv.org/abs/2405.10506, May 2024. doi:10.48550/arXiv.2405.10506.

25 Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960. doi:10.1145/367390.367400.

26 Gaston H. Gonnet, J. Ian Munro, and Derick Wood. Direct dynamic structures for some line segment problems. *Computer Vision, Graphics and Image Processing*, 23(2):178–186, 1983. doi:10.1016/0734-189X(83)90111-1.

27 Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 161–171, 2012. doi:10.1145/2312005.2312036.

28 Prasad Jayanti. f-arrays: implementation and applications. In *Proc. 21st ACM Symposium on Principles of Distributed Computing*, pages 270–279. ACM, 2002. doi:10.1145/571825.571875.

29 Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *Proc. 25th International Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 3821 of *LNCS*, pages 408–419, 2005. doi:10.1007/11590156_33.

30 Donald E. Knuth. *The Art of Computer Programming, Volume 3*. Addison-Wesley, second edition, 1998.

31 Jeremy Ko. A lock-free binary trie. In *Proc. 44th IEEE International Conference on Distributed Computing Systems*, 2024. To appear. Preliminary version available from https://arxiv.org/abs/2405.06208. doi:10.48550/arXiv.2405.06208.

32 Ilya Kokorin, Victor Yudov, Vitaly Aksenov, and Dan Alistarh. Wait-free trees with asymptotically-efficient range queries. In *Proc. IEEE International Parallel and Distributed Processing Symposium*, pages 169–179, 2024. doi:10.1109/IPDPS57955.2024.00023.

33 Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985. doi:10.1137/0214021.

34 Hossein Naderibeni and Eric Ruppert. A wait-free queue with polylogarithmic step complexity. *Distributed Computing*, 2024. Published online August, 2024. doi:10.1007/s00446-024-00471-7.

35 Aravind Natarajan, Arunmoezhi Ramachandran, and Neeraj Mittal. FEAST: a lightweight lock-free concurrent binary search tree. *ACM Transactions on Parallel Computing*, 7(2), May 2020. doi:10.1145/3391438.

36 Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. Bundling linked data structures for linearizable range queries. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 368–384, 2022. doi:10.1145/3503221.3508412.

37 Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proc. 17th ACM SIGPLAN Symposium on*

Principles and Practice of Parallel Programming, pages 151–160, 2012. doi:10.1145/2145816.2145836.

38 Gal Sela and Erez Petrank. Concurrent size. *Proc. of the ACM on Programming Languages*, 6(OOPSLA2):345–372, 2022. doi:10.1145/3563300.

39 Gal Sela and Erez Petrank. Concurrent aggregate queries. Manuscript available from https://arxiv.org/abs/2405.07434, May 2024. doi:10.48550/arXiv.2405.07434.

40 Niloufar Shafiei. Non-blocking Patricia tries with replace operations. *Distributed Computing*, 32(5):423–442, 2019. doi:10.1007/S00446-019-00347-1.

41 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983. doi:10.1145/800076.802464.

42 Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. PAM: parallel augmented maps. In *Proc. 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 290–304, 2018. doi:10.1145/3178487.3178509.

43 Robert Endre Tarjan. *Data Structures and Network Algorithms*, chapter 4.2, pages 45–57. SIAM, Philadelphia, USA, 1983. doi:10.1137/1.9781611970265.

44 Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 31–46, 2021. doi:10.1145/3437801.3441602.

45 Yuanhao Wei, Guy E. Blelloch, Panagiota Fatourou, and Eric Ruppert. Practically and theoretically efficient garbage collection for multiversioning. In *Proc. 28th ACM Annual Symposium on Principles and Practice of Parallel Programming*, pages 66–78, 2023. doi:10.1145/3572848.3577508.

A Pseudocode for Lock-Free Augmented BST

Here, we give more details about how to augment the lock-free BST of Ellen et al. [21]. Type definitions are given in Figure 9. High-level pseudocode for Insert and Delete is in Figure 10. These are mostly the same as in [21], except for the addition of calls to Propagate and the creation of Version objects used to initialize the *version* fields of Nodes created by Insert. Consequently, we do not give all the details of these routines; see [21] for the detailed pseudocode. The new routines for handling Versions and example queries are in Figure 11.

An **Insert**(k) searches for k in the BST of Nodes and arrives at a leaf Node ℓ containing some key k'. If $k' = k$, the value k is already in the BST, so the Insert does not need to modify the tree and will eventually return false. Otherwise, the Insert attempts to replace the leaf ℓ by a new internal Node whose key is $\max(k, k')$ with two new leaf children whose keys are $\min(k, k')$ and $\max(k, k')$. There are also some additional steps required to coordinate updates to the same part of the tree, and those steps may cause the attempt to fail, in which case the Insert tries again by backtracking up the tree and then searching down the tree for the correct place to try inserting the node again. The details of the inter-process coordination are not important to the augmentation. Before attempting to add the three new Nodes to the tree, the Insert creates a new Version object for each of them with fields filled in as shown in Figure 8b. To facilitate backtracking after an unsuccessful attempt, the Insert keeps track of the sequence of internal Nodes visited on the way to the location to perform the insertion in a *thread-local* stack. When an attempt of the Insert succeeds, it calls Propagate on the newly inserted internal Node and returns true. Propagate uses the thread's local stack to revisit the Nodes along the path from the root to the insertion location in reverse order, performing a double Refresh on each Node, as in Section 3. If the Insert terminates after finding the key is already present in a leaf Node, it calls Propagate on that leaf Node, to ensure that the operation that inserted that leaf Node has been linearized, and then returns false.

```
100: type Node                              ▷ used to store nodes of static trie structure
101:     U key                              ▷ immutable key of Node
102:     Node* left, right                  ▷ mutable pointers to children Nodes
103:     Version* version                   ▷ mutable pointer to current Version
104:     Info* info                         ▷ for coordinating updates; irrelevant to augmentation
105: type Version                           ▷ used to store a Node's augmented data
106:     U key                              ▷ immutable key of Node this Version belongs to
107:     Version* left, right               ▷ immutable pointers to children Versions
108:     int sum                            ▷ immutable sum of descendant leaves' bits
```

Figure 9 Object types used in lock-free augmented BST data structure.

A **Delete**(k) has a very similar structure. It first searches for k in the BST of Nodes and arrives at a leaf Node ℓ. If ℓ does not contain k, then the Delete does not need to modify the tree and returns false after calling Propagate. Otherwise, the Delete uses a CAS to attempt to remove both ℓ and its parent from the tree. (See Figure 8c.) Again, there are some additional steps required to coordinate updates to the same part of the tree, which may cause the Delete's attempt to fail and retry, but the details are irrelevant to the augmentation. When an attempt of the Delete succeeds, it calls Propagate to perform a double refresh along a path to the root, starting from the internal Node whose child pointer is changed (i.e., the Node that was formerly the grandparent of the deleted leaf ℓ) and returns true.

Refresh(x) is similar to the routine in Figure 3. Because the structure of the BST's Node tree can change, the repeat loops ensure that the Refresh gets a consistent view of x's child pointer and the contents of that child's *version* field. The other difference is that line 161 stores $x.key$ in the *key* field of the new Version. The **Propagate** routine is identical to the one given in Figure 3, except that we cannot use parent pointers on line 165. Instead, an update operation stores the sequence of Nodes that it traversed from the root to reach a node x and then does a double Refresh on each of them in reverse order (from x to the root).

A query operation is performed on a snapshot of the Version tree obtained by reading *Root.version*. This includes the **Find** operation, which simply performs a search on the Version tree as it would in a sequential BST. As an additional bonus, our Find operation is wait-free, unlike the original lock-free BST [21], where Find operations may starve.

B Sketch of Proof of Correctness for Augmented BST

A detailed proof of linearizability for the augmented BST is in [24]. We sketch it here. As in Section 3.2, we define arrival points of update operations at a Node to indicate when the updates have been propagated to that Node. We linearize updates at their arrival point at the root, and queries when they obtain a snapshot of the Version tree by reading *Root.version*. As in [21], sentinel Nodes as shown in Figure 8a ensure that the root Node never changes.

We again use two main claims: (1) every update operation has an arrival point at the root during the operation, and (2) in every configuration C, the Version tree rooted at a Node x is a legal (augmented) BST containing the set that would result from sequentially performing all operations that have arrival points at x at or before C, in the order of their arrival points. Claim (1) implies the linearization respects the real-time order of operations. Applying Claim (2) to the root shows that queries return results consistent with the linearization.

Although this high-level plan for the proof is similar to Section 3.2, updates' changes to the Node tree introduce some challenges. Firstly, we must ensure that updates are not "lost" if concurrent updates remove the Nodes to which they have propagated. This involves

```
Initialize the data structure as shown in Figure 8a, where Root is a shared pointer
Insert(Key $k$) : Boolean
    let stack be an empty thread-local stack
    push Root on to stack
    loop
        do a BST search for $k$ from top Node on stack, pushing visited internal Nodes on stack
        let $\ell$ be the leaf reached by the search
        if $\ell.key = k$ then
            Propagate(stack)
            return false                          ▷ $k$ is already in the tree
        let $p$ be the top Node $p$ on stack ▷ $p$ was $\ell$'s parent during the search
        let new be a new internal Node whose children are a new leaf Node with key $k$ and a new Leaf with $\ell$'s key. Each of the three new Nodes has a pointer to a new Version object with the same key as the Node. The leaf Versions have sum 1 (or 0 if the key is $\infty_1$ or $\infty_2$) and $new.sum = new.left.sum + new.right.sum$. (See Figure 8b.)
        attempt to change $p$'s child from $\ell$ to new using CAS
        if attempt fails then                     ▷ another update caused failure
            help complete the update that caused the attempt to fail
            backtrack by popping stack until a node that is not marked for deletion is popped, helping complete the deletion of each marked Node that is popped
        else                                      ▷ new was successfully added to tree
            Propagate(stack)
            return true
Delete(Key $k$) : Boolean
    let stack be an empty thread-local stack
    push Root on to stack
    loop
        do a BST search for $k$ from top Node on stack, pushing visited internal Nodes on stack
        let $\ell$ be the leaf reached by the search
        if $\ell.key \neq k$ then
            Propagate(stack)
            return false                          ▷ $k$ is not in the tree
        pop Node $p$ from stack                   ▷ $p$ was $\ell$'s parent during the search
        let gp be the top Node on stack ▷ gp was $p$'s parent during the search
        attempt to change gp's child from $p$ to $\ell$'s sibling using CAS
        if attempt fails then                     ▷ another update caused failure
            help complete the update that caused the attempt to fail
            backtrack by popping stack until a node that is not marked for deletion is popped, helping complete the deletion of each marked Node that is popped
        else                                      ▷ deletion removed $k$'s Node from tree
            Propagate(stack)
            return true
```

Figure 10 Pseudocode for augmented BST. The code for updates is given at a high level. For details, see [21]. Changes to Insert and Delete to support augmentation is shaded.

```
Refresh(Node* $x$) : Boolean                ▷ Try to propagate information to $x$ from its children
    $old \leftarrow x.version$
    repeat                                  ▷ Get a consistent view of $x.left$ and $x.left.version$
        $x_L \leftarrow x.left$
        $v_L \leftarrow x_L.version$
    until $x.left = x_L$
    repeat                                  ▷ Get a consistent view of $x.right$ and $x.right.version$
        $x_R \leftarrow x.right$
        $v_R \leftarrow x_R.version$
    until $x.right = x_R$
    $new \leftarrow$ new Version with $key \leftarrow x.key$, $left \leftarrow v_L$, $right \leftarrow v_R$, $sum \leftarrow v_L.sum + v_R.sum$
    return CAS($x.version$, $old$, $new$)

Propagate(Stack* $stack$)                   ▷ Propagate updates starting at top Node on $stack$
    while $stack$ is not empty do
        pop Node $x$ off of $stack$
        if not Refresh($x$) then
            Refresh($x$)                    ▷ Do a second Refresh if first one fails

Find($k$) : Boolean                         ▷ Returns true if $k$ is in the set, or false otherwise
    $v \leftarrow Root.version$
    while $v.left \neq$ Nil do              ▷ Standard BST search in version tree
        if $k < v.key$ then $v \leftarrow v.left$
        else $v \leftarrow v.right$
    return ($v.key = k$)

Select($j$) : $U$                           ▷ Returns set's $j$th smallest element
    $v \leftarrow Root.version$
    if $j > v.sum$ then                     ▷ Return Nil if size of set is less than $j$
        return Nil
    repeat                                  ▷ Loop invariant: desired element is $j$th in $v$'s subtree
        if $j \leq v.left.sum$ then
            $v \leftarrow v.left$
        else
            $j \leftarrow j - v.left.sum$
            $v \leftarrow v.right$
    until $v$ is a leaf
    return $v.key$

Size : int                                  ▷ Returns number of elements in the set
    return $Root.version.sum$
```

Figure 11 Pseudocode augmented BST, continued. We include Find, Select and Size as three examples of queries that use the augmentation.

transferring arrival points from the removed Node x to another Node x', and this requires proving a number of claims about the arrival points that can be present at x and x' to ensure that transferring arrival points from x to x' does not change the set of keys that should be stored in the Version tree of x'. Secondly, in the original, unaugmented BST of [21], an Insert(k) that reaches a leaf that already contains k returns false, but that leaf may no longer be in the tree when the Insert reaches it, so the linearization point of the Insert is retroactively chosen to be some time during the Insert when that leaf was present in the tree. We must do something similar in choosing the arrival point of failed updates at a leaf.

We describe the arrival points (which in turn defines the linearization) and sketch some of the key arguments about them. For a configuration C, let T_C be the Node tree in configuration C: this is the tree of all Nodes reachable from *Root* by following child pointers. Since our augmentation does not affect the way the Node tree is handled, it follows from [21] that T_C is always a BST. The *search path for key k in C* is the root-to-leaf path in T_C that a BST search for key k would traverse. The following intuition guides our definition of arrival points: the arrival point of an update operation *op* on key k at a Node x should be the first time when both (a) x is on the search path for k and (b) the effect of *op* is reflected in the Version tree rooted at $x.version$. We also ensure that, for any configuration C, the Nodes at which an operation has arrival points defined will be a suffix of the search path for k in C.

A successful Insert(k), shown in Figure 8b, replaces a leaf ℓ containing some key k' by a new internal Node *new* with two children, *newLeaf* containing k, and ℓ', which is a new copy of ℓ. The CAS step that makes this change is the arrival point of the Insert(k) at *new* and *newLeaf*, since these Nodes' Version trees are initialized to contain a leaf with key k. There may also be many operations that had arrival points at ℓ before ℓ is replaced by ℓ' in the Node tree. For example, there may be an Insert(k'') followed by a Delete(k'') if ℓ is the end of the search path for k''. If these operations have not propagated to the root, we must ensure that this happens, so that they are linearized: we do not want to lose the arrival points of these operations when ℓ is removed from the Node tree. So, we transfer all arrival points of update operations at ℓ to *new* and the appropriate child of *new* (depending on whether the key of the update is less than $new.key$ or not).

Similarly, when a Delete(k) changes the Node tree as shown in Figure 8c, each operation with an arrival point at the deleted leaf ℓ (and the Delete(k) itself) is assigned an arrival point at ℓ's sibling *sib*, and at all of *sib*'s descendants on the search path for the operation's key. That operation's key cannot appear in the Version trees of any of those Nodes, so the Version trees of those Nodes correctly reflect the fact that the key has been deleted.

As mentioned above, if an Insert(k) returns false because it finds a leaf ℓ containing k in the tree, [21] proved ℓ *was* on the search path for k in some configuration C during the Insert. Since augmentation has no effect on updates' accesses to the Node, this is still true for the augmented BST. We choose C as the arrival point of the Insert at that leaf. Deletes that return false are handled similarly.

When a Refresh updates the *version* field of a Node x, we assign arrival points to all update operations that had arrival points at x's children before the Refresh read the *version* fields of those children, as in Section 3.2. This indicates that those operations have now propagated to x, and the Version tree in $x.version$ reflects those updates.

We use the definition of arrival points to prove that each update operation's arrival point at the root is between the update's invocation and response. In particular, this reasoning has to argue that no operation gets "lost" as it is being propagated to the root if concurrent deletions remove Nodes to which it has been propagated. Recall that Propagate calls a double Refresh on every Node in the update operation's local *stack*, which remembers all of the internal Nodes visited to reach the leaf ℓ where the update occurs. We use the fact from [21] that Nodes can be removed from the path that leads from the root to ℓ, but no new Nodes can ever be added to it. (It is fairly easy to see that the changes to the Node tree shown in Figure 7 cannot add a new ancestor to ℓ.) Thus, Propagate calls a double Refresh on every ancestor of ℓ to propagate the update all the way to the root Node.

The main invariant says that in every configuration C and in each Node $x \in T_C$, the leaves of the Version tree stored in $x.version$ contain exactly those keys that would be obtained by sequentially performing the operations with arrival points at x at or before C, in the order of their arrival points. Proving this is complicated by the fact that the Node tree changes and arrival points are shifted from one Node to another. We make the argument by focusing on one key k at a time, and showing that k is in the Version tree if and only if the last operation on key k in the sequential execution is an Insert. Moreover, the boolean responses these operations will return are consistent with this sequential ordering. Applying this invariant to the root shows updates return responses consistent with the linearization ordering.

Unlike the trie in Section 3, the subtree rooted at Node x may have a different shape than the Version tree rooted at $x.version$, if updates have changed the Node tree since $x.version$ was stored. However, for any configuration C and any Node $x \in T_C$, the keys of update operations with arrival points at Nodes in the left (or right) subtree of x are less than $x.key$ (or greater than or equal to $x.key$, respectively). Together with the main invariant mentioned above, this allows us to prove that all Version trees are legal BSTs. The correctness of the augmentation fields is trivial, since these fields are correct when an internal Version is created, and its fields are immutable. Hence, queries' results are consistent with the linearization.

Decentralized Distributed Graph Coloring II: Degree+1-Coloring Virtual Graphs

Maxime Flin
Reykjavík University, Iceland

Magnús M. Halldórsson
Reykjavík University, Iceland

Alexandre Nolin
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Abstract

Graph coloring is fundamental to distributed computing. We give the first general treatment of the coloring of virtual graphs, where the graph H to be colored is locally embedded within the communication graph G. Besides generalizing classical distributed graph coloring (where $H = G$), this captures other previously studied settings, including cluster graphs and power graphs.

We find that the complexity of coloring a virtual graph depends linearly on the edge congestion of its embedding. The main question of interest is how fast we can color virtual graphs of constant congestion. We find that, surprisingly, these graphs can be colored nearly as fast as ordinary graphs. Namely, we give a $O(\log^4 \log n)$-round algorithm for the deg+1-coloring problem, where each node is assigned more colors than its degree.

This can be viewed as a case where a distributed graph problem can be solved even when the operation of each node is decentralized.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Mathematics of computing → Graph coloring

Keywords and phrases Graph Coloring, Distributed Algorithms, Virtual Graphs, Congestion, Dilation

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.24

Related Version *Full Version*: https://arxiv.org/abs/2408.11041 [21]

Funding *Maxime Flin*: Icelandic Research Fund (grant 2310015).
Magnús M. Halldórsson: Icelandic Research Fund (grant 217965).

1 Introduction

We give the first full treatment of distributed graph coloring under bandwidth constraints. Namely, we treat the general case when the input graph H differs from the communication graph G. Previously, the problem was studied for cases when $H = G$ (e.g., [54, 7, 41]) or when H has a particular layout in G (e.g., $H = L(G)$ [4, 43], $H = G^2$ [39, 40, 19], or $H = G^k$ [8]).

Most distributed graph algorithms assume that the input graph H is equivalent to the communication network infrastructure G. In the LOCAL model, this is often without loss of generality, as simulating a round of LOCAL on H while communicating on $G = (V_G, E_G)$ without bandwidth restriction is trivial as long as adjacent vertices in H are $O(1)$-hops away in G. When we restrict message size, however, naive simulation is prohibitively inefficient. The delivery of individual messages to each neighbor of a node can slow down the algorithm by a factor proportional to degrees, which might be as high as $n = |V_H|$. Handling cases where $H \neq G$ is an overarching issue in the design of CONGEST algorithms (e.g., in [34, 30, 33, 60, 29, 22, 36, 59, 28]) that is salient when using a CONGEST algorithm

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 24; pp. 24:1–24:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

as a subroutine (e.g., local rounding [15] used in [28]) or when modifying the input graph (e.g., contracting edges [33, 22]). We attempt to study *how bandwidth constraints affect distributed algorithms solving problems on graphs whose description is itself distributed on a communication network.* In this paper, we focus on symmetry breaking and thus ask:

How efficiently can H be colored when distributed on a network G?

Coloring problems are of fundamental importance to distributed graph algorithms. In fact, in its seminal paper [52], Linial studied the locality of 3-coloring cycles. A long line of work [52, 54, 61, 7, 45, 10, 60] showed that $\Delta+1$-coloring could be achieved in $\mathrm{poly}(\log\log n)$ rounds of LOCAL. Further work extended the result to local list sizes [42], and small messages [29, 41, 44]. We extend these results to embedded graphs in nearly the same number of rounds while using local color lists (in a slightly weaker sense than in [42]).

1.1 Virtual Graphs

Before answering our research question, we clarify the meaning of *embedding* a graph H into a network G. We give here a high-level definition and expound on the formal definitions in Section 2. For clarity, we refer to $H = (V_H, E_H)$ as the input or virtual graph while $G = (V_G, E_G)$ is the communication network. We call elements of V_H vertices or nodes while elements of V_G are machines; elements of E_H are edges or conflicts while elements of E_G are links.

We set the definition of embedded virtual graphs forth by specifying which machine knows about which vertex and edge of H. Each vertex $v \in V_H$ is mapped to a set $V(v) \subseteq V_G$ of machines such that *vertices $u, v \in H$ are adjacent (in H) only if their support intersect*, i.e., $V(v) \cap V(u) \neq \emptyset$. We also assume that each support $V(v)$ is equipped with a spanning tree $T(v)$ (called support tree) that can be used to perform aggregation. We assume that machines $w \in V_G$ know about all the supports they belong to – the set of v such that $V(v) \ni w$ – as well as which support tree their adjacent links belong to. Each edge $uv \in E_H$ is mapped to a machine $w \in V(u) \cap V(v)$ in the intersection of the two nodes' supports, which knows about the existence of that edge. Figure 1 exemplifies such an embedding.

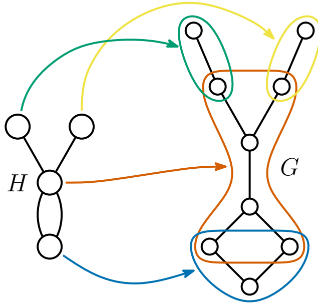


Figure 1 A virtual graph H (on the left) embedded on a network G (on the right). On this example, there is a unique choice of support trees; they have congestion $\mathsf{c} = 1$ and dilation $\mathsf{d} = 3$.

It is convenient to design algorithms for H as a sequence of (virtual) rounds with the same three-step structure[1]: first, broadcast a message to all vertices on the support; second, machines at intersections of supports perform local computations; third, converge-cast the

[1] we emphasize, however, that algorithms are not limited to this scheme and can communicate on the network more cleverly.

result of these computations on the support trees. Naturally, the efficiency of any such algorithm is limited by (1) the diameter of the support trees and (2) the number of trees using the same edge. We call the former the *dilation* and the latter the *congestion*. In some cases, most of the effort is in computing a good embedding, meaning with small enough dilation and congestion. For instance, in [22], the struggle is in finding $n^{o(1)}$-congestion embeddings for various sparsifiers. In this paper, besides direct applications, we assume the embedding is given as part of the input.

Last but not least, we allow H to be a *multi*-graph (without self-loops) to capture the fact that supports can intersect in multiple places. For instance, in Figure 1, the central vertex is adjacent to the bottom vertex through two paths in the network. While distinguishing between the number of incident edges and adjacent vertices is not always necessary, it is crucial for graph coloring, especially when – like in this paper – the number of colors used by each vertex depends on its degree.

1.2 Our Contributions

Our conceptual contribution is an explicit formalization of the notion of *virtual graphs* that captures the aforementioned examples. We show that the key parameters of congestion c and the dilation d essentially capture the hardness of the coloring problem. On one hand, they limit the efficiency of any $\deg + 1$-coloring algorithm:

▶ **Theorem 1.** *Any constant-error algorithm for 3-coloring a 2-regular virtual graph H embedded on a network with bandwidth b, congestion c, and dilation d, requires $\Omega(\frac{\mathsf{c}}{\mathsf{b}} + \mathsf{d} \cdot \log^* n)$ rounds in the worst-case.*

We emphasize that the lower bound applies to algorithms working for any given embedding. It applies to all such algorithms, and not just those following the three-step process described in Section 1.1.

Conversely, we provide a nearly optimal upper bound for coloring virtual graphs. Applied to the CONGEST model – when $H = G$ – its complexity nearly matches the state-of-the-art $O(\log^3 \log n)$ round complexity of [41, 44].

▶ **Theorem 2.** *Let H be a virtual graph on network G with $|V_G| = n$ machines, bandwidth $\mathsf{b} = O(\log n)$, congestion $\mathsf{c} \leqslant n$ and dilation d. There exists an algorithm to $\deg + 1$-color H in $O(\mathsf{cd} \cdot \log^4 \log n)$ rounds. More precisely, at the end of the algorithm, each vertex $v \in V_H$ has a color $\varphi(v) \in \{1, 2, \ldots, \deg(v) + 1\}$ where $\deg(v)$ is the number of edges incident to v in H.*

A key reason for considering the $\deg + 1$-coloring problem is that we forgo using some frequently assumed global knowledge – here, the maximum degree Δ. This is the source of substantial technical challenges, sketched in Section 1.3. That virtual nodes can be connected with multiplicity breaks several classic arguments, hence requires novel ideas to reach the usual goals of providing nodes with excess colors, and classifying them according to their potential in that respect. Our adaptation of the Ghaffari-Kuhn algorithm (see the full version [21, Section 7]) to our distributed paradigm might be of independent interest.

1.3 Technical Overview

The Lower Bound. We prove lower bounds on the congestion and dilation separately. Since a $o(\mathsf{d}\log^* n)$ round algorithm for coloring virtual graphs implies a $o(\log^* n)$ round LOCAL algorithm for coloring cycles, the lower bound on the dilation follows from [52, 57]. To prove the lower bound on the congestion, we provide a probability distribution on gadgets (a 2-regular 16-vertex graph) where vertices are partitioned between two sets V_A and V_B. The gadget is such that if Alice (respectively Bob) knows all vertices and edges incident to V_A (respectively V_B), then for Alice and Bob to assign colors to their vertices such that the coloring is proper, they must communicate $\Omega(1)$ bits. A classic direct sum argument shows that solving k independent copies of this communication problem requires $\Omega(k)$ bits of communication. Finally, we embed the coloring problem on a graph where Alice's vertices are separated from Bob's through a bridge, causing congestion to be $\mathsf{c} = k$.

The Upper Bound: Inaccurate Degrees. The main challenge for coloring virtual graphs is that vertices do not have direct access to their list of available colors (or palette). Previous work [40, 19] demonstrated that it was not necessary if vertices could instead estimate certain local density parameters. While in [39, 19, 20] these density parameters were defined in term of Δ – the *globally known* maximum degree – in this paper, we assume no such global knowledge and aim to use local list sizes; hence, we require a different notion of local sparsity/density. We adapt our definition of embedding to encompass each vertex's local view of its degree. Concretely, we color a multi-graph H where each vertex uses one more color than it has incident edges. We call a vertex *inaccurate* if its number of incident edges is a constant factor larger than its number of adjacent neighbors. Inaccurate vertices require special treatment, for they can skew estimates of local sparsity. Since we use a number of colors dependent on the number of incident edges while each neighbor blocks at most one color, inaccurate vertices are always guaranteed to have an abundance of free colors. After detecting them, we defer coloring inaccurate vertices to the very end of the algorithm.

The Upper Bound: Providing Enough Colors. Every sublogarithmic randomized coloring algorithm [45, 10, 42] has three phases. First, they compute a partial coloring where each vertex has either *low degree* or *many excess colors compared to its uncolored number of neighbors.* Second, they use randomization and symmetry-breaking techniques to take advantage of this excess and color high-degree vertices ultrafast. Third, low-degree vertices are handled fast due to their low degree. In [45, 10, 42], the algorithm produces excess colors by a single-round randomized color trial. When vertices cannot access their palette [3, 19, 18], they resort to approximations that require generating more excess colors in the densest regions on the graph. We follow the same general approach with some major modifications. First, the use of local-list size partially breaks the analysis of slack generation from [41] (and the one of [42] cannot be implemented fast on virtual graphs). Our main technical contribution is to provide sufficient assumptions for a color trial algorithm to generate enough excess colors even when vertices can have small lists (Lemma 11, see [21, Section 5] of the full version). In general, these added assumptions introduce substantial modifications to the accounting of colors throughout the algorithm (see [21, Section 6.1] of the full version).

The Upper Bound: Low-Degree Vertices. Contrary to previous work [40, 19], all high-degree – larger than some $\operatorname{poly}(\log n)$ – vertices are colored with high probability (rather than reducing uncolored degrees to $O(\log n)$). This implies that, for low-degree vertices, colors can be represented using $O(\log \deg) = O(\log\log n)$ bits. The algorithm for coloring low-degree nodes follows the shattering framework of [7]. First, vertices try random colors for $O(\log\log n)$ rounds. This reduces the uncolored parts of the graph to $\mathrm{poly}(\log n)$-sized components. Since nodes do not know their palette, we provide an algorithm for sampling colors likely-enough to succeed. Then, uncolored vertices learn a list of uncolored-degree+1 colors from their palette with an algorithm similar to a binary search. Finally, we simulate the deterministic algorithm of [35] efficiently and complete the coloring. Our main contributions – our algorithms for sampling colors and learning palettes – can be found in Sections 7.1 and 7.2 of the full version [21], respectively.

1.4 Related Work

Distributed coloring has been intensively studied. See, e.g., [52, 6, 61, 7, 46, 24, 45, 10, 56, 39, 35, 41, 42, 25] and references therein. The focus is usually on simple graphs, where the degree refers to the number of neighbors. The state-of-the-art LOCAL algorithm for degree+1-coloring (in terms of n only) is the $\tilde{O}(\log^2\log n)$-round algorithm obtained by plugging the $\tilde{O}(\log^2 n)$-round deterministic algorithm of [27] into the shattering framework of [42]. In CONGEST, authors of [44] show how to implement shattering with small messages; hence, using the $O(\log^3 n)$-round deterministic algorithm of [35], the resulting complexity is $O(\log^3\log n)$. Besides degrees being defined slightly differently, results of [42, 35, 27] are also more general in the sense that vertices can use *any list* of degree+1-colors (not necessarily $\{1, 2, \dots, \deg(v) + 1\}$). Handling less constrained lists of colors in virtual graphs appears out of reach of current techniques; in fact, the problem has yet to be tackled in the simpler settings of cluster graphs and power graphs.

Virtual Graphs. Virtual graphs are ubiquitous in distributed graph algorithms and we make no attempt to be exhaustive. They refer to cases where the input graph differs from the communication network, though the formalism varies by use case. Here, we list occurrences of greatest relevance.

1. Many algorithms modify the input graph – e.g., by contracting an edge or removing a vertex and adding an edge between each neighbor – throughout the execution. This happens, e.g., in [33, 22]. In such cases, the algorithms embed the modified graph into the network while ensuring low congestion. Authors of [30, 59, 2] show that under some assumptions on the graph (e.g., planarity or excluded minor) then low-congestion shortcuts can be found efficiently, leading to drastic improvements on the round complexity.
2. Recent network decomposition algorithms [60, 29, 28] compute clusters – i.e., sets of vertices – by growing increasingly large sets of vertices. Hence, computations are held through the three-step aggregation process described in Section 1.1. That is, these algorithms are computing sequences of virtual graphs (including support trees) with $\mathrm{poly}(\log n)$ dilation and congestion.
3. Finally, the local rounding framework introduced in [16] and perfected in [15] runs a defective-coloring subroutine on virtual graphs. They describe d2-multigraphs, a special case of virtual graphs used to implement their algorithm in CONGEST. They care for parallel edges since they compute a coloring, like us. Besides, their rounding algorithm has been used by network decomposition algorithms [28, 27] and thus had to be implemented on virtual graphs.

Our formalism for virtual graph captures all mentioned examples (with & without congestion, with & without parallel edges).

Scheduling & Routing. Congestion and dilation are natural parameters in routing problems, where they measure the maximum overlap and length of the delivery paths of a set of packets. Scheduling, in this context, refers to organizing the packets' delivery along their paths, taking into account congestion constraints. Naive scheduling leads to a $O(\mathsf{cd})$ delivery time, which can be hard to improve upon distributedly. Asymptotically optimal $\Theta(\mathsf{c}+\mathsf{d})$ schedules exist and can be computed efficiently given global knowledge of the paths [49, 50].

The routing literature is expansive and growing to this day [48, 51, 26, 32, 38]. While parallel delivery of information is crucial to our virtual graph algorithms, our problems are quite distinct from typical routing questions, as we usually aggregate and broadcast information rather than deliver it from a single source to a single target. In particular, we often change the information during its delivery. Even for our more complex tasks, a naive scheduling in $O(\mathsf{cd})$ remains possible. We leave open the question of whether the $O(\mathsf{cd})$ dependency can be improved to $O(\mathsf{c}+\mathsf{d})$ (see Problem 3 in Section 5 for more).

Power Graphs. Recently, there has been a growing interest in bandwidth-efficient algorithms for power graphs [39, 40, 5, 55, 19, 8]. Theorem 2 improves on previous work about distance-2 coloring [39, 40, 19] by handling a more general problem (see Section 2.1), by reducing the number of colors used by each vertex to its pseudo-degree (rather than, say, using Δ^2+1 colors which depends on a global parameter), and by improving the runtime by several $O(\log\log n)$ factors.

Other Models. The Congested Clique [53] can be viewed as a virtual graph model on the opposite end of the spectrum, where the communication graph is a clique. It has a $O(1)$-round deterministic algorithm for $\deg+1$-list-coloring [11], building on similar results for $\Delta+1$-coloring [9, 12].

Sibling Paper. In a sibling paper [20], we treat cluster graphs, a particular type of virtual graphs, focusing on high-degree graphs. We give a $O(\log^* n)$-round algorithm for $\Delta+1$-coloring cluster graphs when $\Delta=\Omega(\log^{21} n)$. A key technical contribution is coloring so-called put-aside sets in extremely dense subgraphs, which we build on in this paper. That paper introduces essential primitives that apply to general virtual graphs, particularly operations on the communication backbone, including broadcast, aggregation, and palette queries. It also contains a fingerprinting technique for approximating the sizes of neighborhoods.

1.5 Outline of Paper

In the next section, we describe the modeling of virtual graphs and show how they capture two important settings. We present the main ideas behind the lower bound in Section 3. The high-level view of the algorithm is given in Section 4.3 along with key definitions, before describing some open questions in Section 5.

The detailed descriptions of various parts of the algorithm are deferred to the full version of this paper [21]. In [21, Section 5] we give a result on slack generation, generalizing previous arguments to $deg+1$-colorings (of both sparse and dense nodes). The coloring of different parts of the graph is split into several sections: the dense-but-not-too-dense part in [21, Section 6], the low-degree nodes in [21, Section 7], while the extremely-dense are in [21, Appendix C] as it builds heavily on the sibling paper [20]. The details of the lower bound are in [21, Appendix D]. Further appendices feature various algorithmic steps that are non-trivial adaptations or modifications of previous work, including almost-clique decomposition in [21, Appendix F].

1.6 Preliminaries

Mathematical Notation. For an integer $t \geqslant 1$, let $[t] \overset{\text{def}}{=} \{1, 2, \ldots, t\}$. For a function $f : \mathcal{X} \to \mathcal{Y}$, when $X \subseteq \mathcal{X}$, we write $f(X) \overset{\text{def}}{=} \{f(x) : x \in X\}$; and when $Y \subseteq \mathcal{Y}$, we write $f^{-1}(Y) \overset{\text{def}}{=} \{x \in X : f(x) \in Y\}$. We abuse notation and write $f^{-1}(y) \overset{\text{def}}{=} f^{-1}(\{y\})$. For $X \subseteq \mathcal{X}$, we write $f_{|X} : X \to \mathcal{Y}$ for the restriction of f to X. Throughout the paper, we hide overhead due to congestion c and dilation d by writing $\widehat{O}(f)$ for $O(\mathsf{cd} \cdot f)$.

Graphs & Multi-Graphs. A multi-graph $H = (V_H, E_H)$ is defined by a set of vertices V_H and sets $E_H(u, v)$ describing all edges between u and v (and $E_H(u, v) = \emptyset$ if u and v are not adjacent). When each set $E_H(u, v)$ contains at most one edge (H has no parallel edges), we say the graph is ***simple***. The neighbors of v in H are $N_H(v) \overset{\text{def}}{=} \{u \in V_H : E_H(u, v) \neq \emptyset\}$. The ***pseudo-degree*** of v in H is $\deg(v; H) \overset{\text{def}}{=} \sum_{u \in V_H} |E_H(u, v)|$, its number of incident edges. Its ***degree*** counts its neighbors $|N_H(v)|$. When H is clear from context, we drop the subscript and write $N(v) = N_H(v)$ and $\deg(v) = \deg(v; H)$. An unordered pair $\{u, v\} \subseteq V_H$ is called an ***anti-edge*** or ***non-edge*** if $E_H(u, v) = \emptyset$.

Colorings. For any integer $q \geqslant 1$, a ***partial*** q***-coloring*** is a function $\varphi : V_H \to [q] \cup \{\bot\}$ where $\bot$ means “not colored”. The domain $\mathsf{dom}\,\varphi \overset{\text{def}}{=} \{v \in V_H : \varphi(v) \neq \bot\}$ of φ is the set of colored nodes. A coloring φ is ***total*** when all nodes are colored, i.e., $\mathsf{dom}\,\varphi = V_H$; and we say it is ***proper*** if $\bot \in \varphi(\{u, v\})$ or $\varphi(v) \neq \varphi(u)$ whenever $E_H(u, v) \neq \emptyset$. We write that $\psi \succeq \varphi$ when a partial coloring ψ ***extends*** φ: for all $v \in \mathsf{dom}\,\varphi$, we have $\psi(v) = \varphi(v)$. The ***uncolored degree*** $|N_\varphi(v)| \overset{\text{def}}{=} |N(v) \setminus \mathsf{dom}\,\varphi|$ of v with respect to φ is the number of uncolored neighbors of v. The ***uncolored pseudo-degree*** $\deg_\varphi(v)$ of v with respect to φ counts its number of incident edges to uncolored neighbors. The ***palette*** of v with respect to a coloring φ is $L_\varphi(v) = [\deg(v) + 1] \setminus \varphi(N(v))$, the set of colors we can use to extend φ at v.

2 Virtual Graphs

In distributed algorithmics, we consider ***communication graphs*** or ***networks*** $G = (V_G, E_G)$ where elements of V_G are ***machines*** that communicate by sending messages on incident ***links*** – unordered pairs of E_G – simultaneously in synchronous rounds. We assume machine $v \in V_G$ is provided a $O(\log |V_G|)$-bits unique identifiers ID_v to break symmetry. For randomized algorithms, they can also access local random bits. Messages are limited to b bits, where b is called the ***bandwidth*** of the network. Unless stated explicitly, it is assumed that $\mathsf{b} = \Theta(\log |V_G|)$.

We define our notion of virtual graphs formally. We shall always refer to the conflict graph by H and to the communication graph by G. Vertices/nodes and edges refer only to elements of H, while machines and links are used for G.

▶ **Definition 3** (Virtual Graph). *Let $G = (V_G, E_G)$ be a* simple *graph. A virtual graph on G is a* multi-*graph $H = (V_H, E_H)$ where each vertex $v \in V_H$ is mapped to a set $V(v) \subseteq V_G$ of machines called the* **support** *of v. Whenever two nodes are adjacent in H their supports intersect, i.e., if $E_H(u, v) \neq \emptyset$ then $V(u) \cap V(v) \neq \emptyset$. Each machine $w \in V_G$ knows the set $V^{-1}(w)$ of vertices whose supports contains it.*

When bandwidth is not an issue, we can work directly with the representation of Definition 3. We can compute a breadth-first spanning tree $T(v) \subseteq E_G$ on each support $V(v)$ for distributing information, and then simulate a local algorithm on this support structure. With bandwidth constraints, we need to be more careful.

▶ **Definition 4** (Embedded Virtual Graph). *Let H be a virtual graph on G such that $|V_H| \leqslant \text{poly}(|V_G|)$. Suppose that (1) for each vertex $v \in V_H$, there is a tree $T(v) \subseteq E_G$ spanning $V(v)$; and (2) for each edge $e \in E_H(u,v)$ there is a machine $m(e) \in V(u) \cap V(v)$. We call $T(v)$ the* **support tree** *of v and $m(e)$ the machine* **handling** *edge e. Each machine w knows the set of edges $m^{-1}(w)$ it handles as well as, for each incident link $\{w,w'\} \in E_G$, the set $T^{-1}(ww')$ of support trees it belongs to.*

Given support trees, it is convenient to design our algorithms as a sequence of rounds each consisting of a three-step process: broadcast, local computation on edges, followed by converge-cast. We use two parameters to quantify the overhead cost of aggregation on support trees. The ***congestion*** c of H is the maximum number of trees using the same link. The ***dilation*** d is the maximum height of a tree $T(v)$ in G. Formally,

$$\mathsf{c} \overset{\text{def}}{=} \max_{e \in E_G} |T^{-1}(e)| \quad \text{and} \quad \mathsf{d} \overset{\text{def}}{=} \max_{v \in V_H} \left(\max_{u \in T(v)} \text{dist}_{T(v)}(v,u) \right) . \tag{1}$$

Congestion and dilation are natural bottlenecks for virtual graphs. In Theorem 1, we show that $\Omega(\mathsf{c}/\mathsf{b} + \mathsf{d} \log^* n)$ rounds are needed for our coloring task given b bandwidth in the communication graph. Conversely, Theorem 2 shows that coloring in $O(\mathsf{cd} \cdot \log^4 \log n)$ rounds is feasible for any embedding.

▶ Remark 5. A few remarks are in order.

1. The degrees in H can be computed as $\deg(v) = \sum_{w \in T(v)} |m^{-1}(w)|$ by aggregation on support trees, which is why we ask that edges have designated handlers. Counting exactly the number of distinct neighbors for all vertices appears to be challenging (i.e., requires $\widetilde{\Omega}(|N_H(v)|)$ rounds).
2. By running a BFS from a single source (or from multiple sources but in *disjoint* subgraphs), we can count the exact the number of neighbors the source has. However, running this algorithm from multiple vertices creates congestion proportional to that number of vertices.
3. It is, per se, easy to compute low-diameter support trees for all vertices, e.g., by BFS, but a poor selection of edges could easily lead to high congestion. It is an open question if trees of both low diameter and congestion can be computed efficiently (see Section 5).

2.1 Implications

Our framework captures several models and problems studied in the distributed graph literature. We review them quickly.

It is helpful to see the communication network $G = (V_G, E_G)$ through its ***subdivision graph***: the bipartite graph $S_G = (V_G, E_G, \{\{u,e\} : u \in e \in E_G\})$ with machines on the left, links on the right, and a link between $v \in V_G$ and $e \in E_G$ if and only if v is an endpoint of e. Simulating a round of communication on S_G takes one round of communication of G (conversely, one round on G takes two rounds of S_G). Defining our virtual graphs on S_G rather than G allows us to put conflict on links. We call the links of S_G ***half-links***. [2]

[2] A common alternative representation is to stipulate that edges are between vertices with adjacent supports, i.e., $uv \in E_H$ implies that $\exists w \in V(v), x \in V(u)$ s.t. $wx \in E_G$. If we extend each support $V(v)$ in G to include also the incident link nodes in S_G, then two supports in S_G intersect whenever they are adjacent in G. Hence, our formulation encompasses this variant.

2.1.1 Application 1: Cluster Graphs

A ***cluster graph*** $\mathcal{C}$ on a communication graph $G = (V_G, E_G)$ is a graph where vertices are disjoint sets $C_x \subseteq V_G$ called ***clusters*** with a designated machine $\mathsf{leader}(x) \in C_x$. Each cluster C_x induces a connected graph of small diameter in G. Two clusters are adjacent if and only if they are connected by a link. A round of communication on H consists of 1) broadcasting a b-bit message from $\mathsf{leader}(x)$ to all machines in C_x, 2) communication on inter-cluster links, and 3) aggregate a $\operatorname{poly}\log n$-bit message (e.g., a sum or a min) to $\mathsf{leader}(x)$. They appear in several places, from maximum flow algorithms [33, 22] to network decomposition [60, 29].

Clearly, cluster graphs are captured by our definition of virtual graphs: for $C_x \in V_H$, let $V(C_x)$ be C_x plus the half-links going out of C_x and $T(C_x)$ be a BFS tree spanning $V(C_x)$. Theorem 2 implies we can color cluster graphs fast:

▶ **Corollary 6.** *Cluster graphs can be* $\deg +1$*-colored, w.h.p., in* $O(\mathsf{d} \cdot \log^4 \log n)$ CONGEST *rounds where* d *is the maximum (strong) diameter of a cluster, i.e., of* $H[C_x]$ *over all* C_x*.*

In [20], we show that $\Delta + 1$-coloring high-degree cluster graphs (where $\Delta \leqslant \operatorname{poly}(\log n)$) can be done in $O(\log^* n)$ rounds. Corollary 6 is the first non-trivial distributed algorithm for degree+1-coloring cluster graphs.

2.1.2 Application 2: Coloring Power Graphs

For some integer $t \geqslant 1$, the ***t-th power graph*** of G is the graph G^t on vertices V_G where there is an edge $\{u, v\}$ when $\operatorname{dist}_G(u, v) \leqslant t$. A ***distance-t coloring*** of G is a coloring of G^t. Concretely, it is a coloring where nodes receive colors different from the ones in their t-hop neighborhood. Our framework provides a unified view of distance-t colorings: the same algorithm provides fast algorithms for all values of $t \geqslant 1$.

▶ **Lemma 7.** *Let $t \geqslant 1$ and $G = (V_G, E_G)$ be a graph with maximum (distance-1) degree Δ. We can define a virtual graph $H = (V_H, E_H)$ on the subdivision graph S_G of G such that $V_H = V_G$ and a* $\deg +1$*-coloring of H is a* $\Delta^t + 1$*-coloring of G^t. Moreover, the congestion is* $\mathsf{c} = O(\Delta^{\lfloor \frac{t-1}{2} \rfloor})$*, the dilation is* $\mathsf{d} = t$*, and the embedding is computable in* $O(t\mathsf{c})$ *rounds.*

Proof. For each node $v \in V_G$, its support tree $T(v)$ in G is set to be the *t-hop BFS tree in the subdivision graph S_G rooted at v*. For any pair $u, v \in V_H$, the edge set $E_H(u, v) = \emptyset$ if and only if $\operatorname{dist}_G(u, v) > t$. Otherwise, $E_H(u, v)$ contains an edge for each *simple uv-paths* in $T(u) \cup T(v)$ in G. As there are at most $\sum_{i=1}^{t-1} \Delta(\Delta - 1)^i \leqslant \Delta^t$ simple paths of length at most t starting from v in G. Hence, each vertex is incident to at most Δ^t edges in H. Thus, any $\deg +1$-coloring on H is a distance-t coloring of G with $\Delta^t + 1$ colors and from the definition of edges in H, a proper coloring on H is also proper on G^t.

The bound on the dilation is immediate. To verify the congestion on a half-link ev, observe that there are at most $\Delta^{\lfloor \frac{t-1}{2} \rfloor}$ nodes (of G) that are within distance t of v in S_G, and therefore at most that many support trees using that half-link.

We map each simple uv-path in $T(u) \cup T(v)$ to its middle machine in S_G. It is unique, as S_G is bipartite and u, v are on the same side. Each machine $w \in T(u) \cap T(v)$ knows its distances to u in $T(u)$ and to v in $T(v)$, and thereby knows if it is the middle machine. To compute the embedding, we have each machine learn its distance-t neighborhood in S_G, with the distance it has to each machine in it. This is done as follows: initially, each machine v prepares a message $(\mathsf{ID}_v, 1)$, which it sends to its Δ direct neighbors in G. Then, for each positive integer $i \leqslant \lfloor \frac{t-1}{2} \rfloor$, each machine sends a message of the form $(\mathsf{ID}_u, i + 1)$ to its direct neighbors for each message (ID_u, i) it has received, of which there are at most Δ^i. Sending all messages for a fixed i takes $O(\Delta^{\lfloor \frac{t-1}{2} \rfloor}) = O(\mathsf{c})$ rounds, hence a total $O(t\mathsf{c})$ complexity. At the end of this process, each machine v knows to which support trees $T(u)$ it belongs, and for each simple path of length at most t in S_G between two nodes u, u' s.t. $v \in T(u) \cap T(u')$, v knows whether it is its midpoint and should thus handle the edge. ◀

For any $t \geqslant 1$, Theorem 2 and Lemma 7 imply that there is a distributed algorithm communicating on S_G with $O(\log n)$ bandwidth that computes a $\Delta^t + 1$-coloring of G^t. Since a round of communication on S_G can be simulated in one round of communication on G, it shows the following corollary.

▶ **Corollary 8.** *For any $t \geqslant 1$, there is a randomized* CONGEST *algorithm for $\Delta^t + 1$-coloring G^t that runs in $O(t\Delta^{\lfloor (t-1)/2 \rfloor} \log^4 \log n)$ rounds w.h.p.*

Furthermore, the specific structure of power graphs allows for broadcast and aggregation over support trees to be done in only $O(\Delta^{\lfloor (t-1)/2 \rfloor}) = O(\mathsf{c} + \mathsf{d})$ rounds instead of $O(\mathsf{cd}) = O(t\Delta^{\lfloor (t-1)/2 \rfloor})$. The runtime in Corollary 8 can be improved to $O(\Delta^{\lfloor (t-1)/2 \rfloor} \log^4 \log n)$ as a result.

It is not too difficult to see that – by a reduction to set disjointness – verifying an *arbitrary (or random)* distance-t coloring needs $\tilde{\Omega}(\Delta^{\lfloor \frac{t-1}{2} \rfloor})$ rounds in CONGEST [23]. However, no super-constant lower bounds are known for computing distance-t colorings in CONGEST when $t \geqslant 3$ and $\Delta \gg \log n$.

3 Lower Bounds: Overview

In this section, we sketch the main ideas behind our lower bound. We show the following theorem:

▶ **Theorem 1.** *Any constant-error algorithm for 3-coloring a 2-regular virtual graph H embedded on a network with bandwidth b, congestion c, and dilation d, requires $\Omega(\frac{\mathsf{c}}{\mathsf{b}} + \mathsf{d} \cdot \log^* n)$ rounds in the worst-case.*

This implies as immediate corollary the same lower bound for the more general problem of deg +1-coloring virtual graphs. The single statement is actually the concatenation of two independent lower bounds, one relative to congestion and bandwidth, and the other relative to dilation.

The dilation lower bound is straightforward, following directly from the seminal $\Omega(\log^* n)$ lower bounds on 3-coloring [52, 57]. We refer readers to Appendix D.2 of the full version [21].

As the congestion lower bound makes lengthy use of technical tooling from communication complexity literature largely unrelated to the rest of the paper, we defer most details to Appendix D of the full version [21]. Here, we chiefly describe the virtual graphs used for our lower bound and give intuition behind the complexity of coloring them.

Proof Structure of the Congestion Lower Bound. The congestion-related part of our lower bound is obtained through a reduction from communication complexity. Our overall proof plan is as follows:

- We introduce a 2-player communication complexity task in which said players must coordinate to 3-color a 16-node 2-regular graph. Each player only knows the edges incident to half of the vertices and is in charge of outputting half of the coloring.
- We show that this task is nontrivial, and in particular, that it has $\Omega(1)$ information complexity, a complexity measure which lower bounds communication complexity.

- From known direct-sum results on information complexity, we get that solving $\mathsf{c}/8$ independent copies of the task has information complexity $\Omega(\mathsf{c})$.
- We introduce a virtual graph of congestion c and constant dilation in which we embed $\mathsf{c}/8$ instances of the task, i.e., deg +1-coloring the virtual graph solves the $\mathsf{c}/8$ instances.
- We observe: any T-round algorithm for deg +1-coloring virtual graphs over communication graphs with congestion c given bandwidth b implies a $O(T\mathsf{b})$ communication complexity algorithm for solving $\mathsf{c}/8$ copies of the nontrivial task.
- We conclude: the round complexity T of any such distributed algorithm for deg +1-coloring must necessarily be at least $\Omega(\mathsf{c}/\mathsf{b})$.

The Communication Complexity Gadget. We define the communication complexity task we use in Definition 9. While this definition is a generalization with an arbitrary even number of nodes on both sides, for our purposes, we will only use the gadget with the parameter k set to $k = 4$, i.e., with 8 nodes on Alice and Bob's sides. See Figure 2 for a illustration of our gadget.

▶ **Definition 9** (Matching 3-coloring task). *In the* $\mathsf{M3COL}_k$ *task, a $4k$-node graph is initially uncolored. Its nodes are split into two equal parts – left and right – given to Alice and Bob. Alice and Bob receive a perfect matching over their respective sets of $2k$ nodes. For each $i \in [2k]$, the ith left node is connected to the ith right node. At the end of the communication protocol, Alice must output a color in $\{1, 2, 3\}$ for each left node, and Bob must do the same for the right nodes, such that the coloring is valid with respect to the graph received as input.*

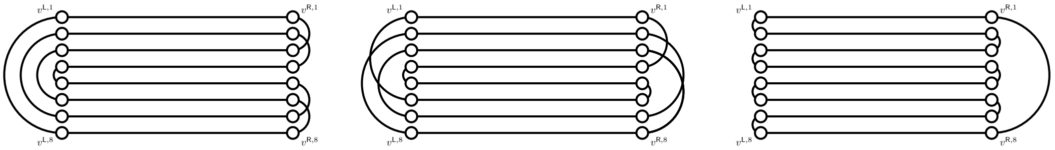


Figure 2 Three possible inputs to the communication complexity task.

The crux of the argument is to show that the $\mathsf{M3COL}_4$ task cannot be solved without communication. This can be intuitively seen by noticing that there can be at most 3 nodes on which Alice *always* outputs the same color regardless of her input matching (same on Bob's side). Indeed, as there are only 3 colors, a fourth node with a fixed color means that two nodes would receive the same color on Alice's side regardless of her matching. This implies an error when said two nodes are connected in Alice's matching. Generalizing this idea to randomized algorithms allows us to show that an algorithm without communication necessarily makes an error with some constant probability [3] .

▶ **Lemma 10.** *Any zero communication protocol for* $\mathsf{M3COL}_4$ *fails with probability at least* $\frac{1}{196}$ *over the uniform input distribution.*

Embedding the Gadget. Embedding the gadgets into a virtual graph is then done with the following communication network: we consider two stars (depth-1 trees) with c leaves; we connect the two stars by a single link between their roots $w^{\mathsf{L},\mathsf{com}}$ and $w^{\mathsf{R},\mathsf{com}}$. The support of each node on the left is made of an edge of the left star with the central edge, while the

[3] This intuition also explains why we take gadgets with 8 nodes on each side and not less: a smaller gadget would be solvable without communication by fixing the color of (up to) 3 nodes on each side.

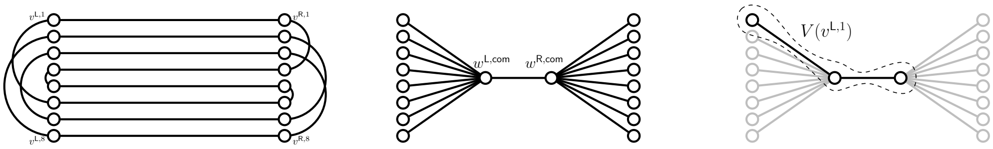


Figure 3 Examples of a virtual graph H with a single gadget (left), a communication network G (middle) in which H can be embedded, and the support of the top left virtual node (right).

support of each node on the right is just an edge in the right star. $w^{\mathsf{L,com}}$ handles the edges in the left matching, while $w^{\mathsf{R,com}}$ handles the edges of the right matching as well as the edges between the left and right sides of the virtual graph. See Figure 3 for an illustration.

The proof of Lemma 10, with its implication for the information complexity of the task, and ultimately, our $\Omega(\mathsf{c}/\mathsf{b})$ lower bound for 3-coloring graphs of degree 2 (Theorem 1), are all deferred to Appendix D.1 of the full version [21].

4 Coloring Algorithm

The goal of this section is to present the main technical ideas behind Theorem 2.

▶ **Theorem 2.** *Let H be a virtual graph on network G with $|V_G| = n$ machines, bandwidth $\mathsf{b} = O(\log n)$, congestion $\mathsf{c} \leqslant n$ and dilation d. There exists an algorithm to* $\deg +1$*-color H in $O(\mathsf{cd} \cdot \log^4 \log n)$ rounds. More precisely, at the end of the algorithm, each vertex $v \in V_H$ has a color $\varphi(v) \in \{1, 2, \dots, \deg(v) + 1\}$ where $\deg(v)$ is the number of edges incident to v in H.*

We give necessary definitions and self-contained statements of each of the main steps of our algorithm. First, we discuss the concept of *slack* and present the means by which we measure and produce it. We then introduce a version of the sparse-dense decomposition tailored to our needs. Finally, we describe the main steps of our algorithm. For more details on individual steps of the algorithm, we refer readers to the full version of this paper [21].

4.1 Slack

Intuitively, the slack measures how easily a vertex gets colored. More formally, it is used to bound from below the number of colors available to a vertex when its neighbors are trying to get colored. There are several types of slack that occur.

Savings. Whenever a neighbor uses a color that is either outside v's palette or the same color as another neighbor, then v *saves* a color. Under a given partial coloring φ, this is quantified by the ***savings slack*** of v from coloring φ:

$$\xi_\varphi(v, S) \overset{\text{def}}{=} |S \cap \mathsf{dom}\,\varphi| - |\varphi(S) \cap [\deg(v) + 1]| \tag{2}$$

We write $\xi_\varphi(v)$ for $\xi_\varphi(v, N(v))$.

Redundancy. In degree+1-coloring (unlike $\Delta + 1$-coloring), slack can also occur when v has a shortage of neighbors with a high enough degree. We measure this with the ***redundancy*** of v defined as

$$\rho_v \overset{\text{def}}{=} \max_{t \leqslant |N(v)|/12} |N_H(v)| - t - |\{u \in N(v) : \deg(u) + 1 > t\}| \ . \tag{3}$$

In other words, there is a $t \leqslant |N(v)|/12$ such that, even if all high-degree neighbors (larger than t) use different colors, at least ρ_v colors remain available to v.

Inaccuracy. The difference between the palette size and the number of neighbors is the ***inaccuracy*** of the node:

$$\delta_v = \deg(v) - |N_H(v)| \ . \tag{4}$$

In our setting, this is caused by parallel edges. A vertex with $\delta_v > \delta$ is called ***δ-inaccurate*** and ***δ-accurate*** otherwise.

Permanent & Temporary slack. The aforementioned forms of slack (savings, redundancy, and inaccuracy) are ***permanent***, meaning that they do not decrease as we extend the coloring. Another way to provide slack to a vertex is by keeping some of its uncolored neighbors inactive. This artificially reduces degrees – thus contention – without reducing the number of available colors, thereby providing slack. This is called ***temporary slack*** as it perishes when we eventually color the inactive neighbors.

Slack Generation. While redundancy and inaccuracies do not depend on the coloring, vertices get savings only if we manage to same-color its neighbors. We show in [21, Section 5] that a classic one-round algorithm of "trying a random color" creates enough slack for deg+1-colorings. This generalizes results for $\Delta+1$-coloring [58, 14, 41]. It also generalizes a method of [1, Lemma 4.10] for deg+1-coloring that applies to the sparse and uneven nodes (assuming $\deg(v) = |N_H(v)|$). `SlackGeneration` creates color savings probabilistically. The savings expected from a random color trial are measured by the unevenness and sparsity, which we now define.

The savings we expect due to high-degree neighbors using colors beyond $\deg(v)+1$ is captured by the ***unevenness*** of v. Within a subgraph induced by a set $S \subseteq V_H$, it is defined as

$$\eta_v(S) \overset{\text{def}}{=} \sum_{u\in S} \frac{[\deg(u)+1] \setminus [\deg(v)+1]}{[\deg(u)+1]} = \sum_{u\in S} \frac{(\deg(u)-\deg(v))^+}{\deg(u)+1} \ . \tag{5}$$

We write $\eta_v = \eta_v(N(v))$ for succinctness. A vertex such that $\eta_v > \eta$ is called ***η-uneven*** and ***η-balanced*** otherwise.

The savings we expect from colors reused by multiple neighbors is quantified by the ***sparsity*** of v. The sparsity of v is defined as

$$\zeta_v \overset{\text{def}}{=} \frac{1}{|N_H(v)|}\left(\binom{|N_H(v)|}{2} - \frac{1}{2}\sum_{u\in N_H(v)} |N_H(u) \cap N_H(v)|\right) \ . \tag{6}$$

Note that $\frac{1}{2}\sum_{u\in N_H(v)}|N_H(u) \cap N_H(v)|$ counts the number of edges in $N_H(v)$ *without multiplicity*, even if H is not simple. Hence $\zeta_v \cdot |N_H(v)|$ counts the number of edges missing in $N_H(v)$, without multiplicity. A vertex such that $\zeta_v > \zeta$ is called ***ζ-sparse*** and ***ζ-dense*** otherwise.

▶ **Lemma 11** (Slack Generation). *Let $V_{\mathsf{sg}} \subseteq V_H$ and let φ_{sg} be the coloring produced by running Algorithm 2 in $H[V_{\mathsf{sg}}]$ avoiding colors $\leqslant r$. Let $v \in V_{\mathsf{sg}}$ be a node satisfying $\deg(v) \leqslant 3|N_H(v)|/2$, $|N(v) \setminus V_{\mathsf{sg}}| < (\zeta_v + \eta_v)/4$, $\zeta_v \geqslant 48r$, and $\rho_v \leqslant (\zeta_v+\eta_v)/12$. Then*

$$\xi_{\varphi_{\mathsf{sg}}}(v) \geqslant \gamma_{11} \cdot (\zeta_v + \eta_v) \quad \textit{with probability} \quad 1 - \exp(-\Theta(\zeta_v+\eta_v))$$

4.2 Almost-Clique Decomposition

In Lemma 12, we describe a structural decomposition partitioning vertices according to their ability to receive slack and of which type. All sub-logarithmic distributed coloring algorithms [45, 10, 42, 20] use such a decomposition. We adapt [1] to account for inaccuracies in degrees (Property 2). Lemma 12 partitions vertices into high- and low-degree vertices based on the threshold $\Delta_{\mathsf{low}} = \Theta(\log^{21} n)$. Each requires a different approach and, in particular, we do not need to argue that low-degree vertices receive slack. We prove Lemma 12 in [21, Appendix F] to preserve the flow of the paper.

▶ **Lemma 12.** *There exists an algorithm that, for any multi-graph $H = (V_H, E_H)$ and $\varepsilon \in (0, 1/100)$, computes in $\widehat{O}(1/\varepsilon^6)$ rounds an ε-***almost-clique decomposition***: a partition $V_H = V_{\mathsf{low}} \cup V_{\mathsf{high}}$ and $V_{\mathsf{high}} = V_{\mathsf{in}} \cup V_\star \cup V_{\mathsf{dense}}$ such that*

1. *each $v \in V_{\mathsf{low}}$ has $\deg(v) \leqslant 2\Delta_{\mathsf{low}}$ and $v \in V_{\mathsf{high}}$ has $\deg(v) \geqslant \Delta_{\mathsf{low}}$;*
2. *each $v \in V_{\mathsf{in}}$ is $\Omega(\varepsilon^3|N(v)|)$-inaccurate and each $v \in V_{\mathsf{high}} \setminus V_{\mathsf{in}}$ has $\deg(v) \leqslant (1+\varepsilon^3)|N(v)|$;*
3. *each $v \in V_\star$ has $\zeta_v + \eta_v + |N(v) \cap V_{\mathsf{in}}| \geqslant \gamma_{12} \cdot \deg(v)$ for a constant $\gamma_{12} = \gamma_{12}(\varepsilon) \in (0, 1)$;*
4. *V_{dense} is partitioned into ε-almost-cliques: sets $K \subseteq V_{\mathsf{high}}$ such that*
   a. *$|N_H(v) \cap K| \geqslant (1-\varepsilon)|K|$, for each $v \in K$,*
   b. *$\deg(v) \leqslant (1+\varepsilon)|K|$, for each $v \in K$, and*
   c. *$|N_H(v) \setminus K| \leqslant O_\varepsilon(\zeta_v + \eta_v + |N(v) \cap V_{\mathsf{in}}|)$.*

Let $\Delta_K \overset{\text{def}}{=} \max_{v \in K} \deg(v)$. From Lemma 12, it holds for each almost-clique K that $(1-\varepsilon)|K| \leqslant \Delta_K \leqslant (1+\varepsilon)|K|$, and that for every $v \in K$, $\deg(v) \geqslant (1-2\varepsilon)\Delta_K$. Every pair of vertices in K has $(1-2\varepsilon)|K|$ neighbors in common in K, and hence $H[K]$ has (strong-)diameter at most two.

For $v \in V_{\mathsf{dense}}$, let K_v denote the almost-clique containing v. We denote by $A_v = K_v \setminus N_H(v)$ its ***anti-neighborhood*** and by $a_v = |K_v| - \deg(v, H \cap K_v)$ its ***pseudo-anti-degree***. We call $E_v = N_H(v) \setminus K_v$ the ***external-neighborhood*** and $e_v = \deg(v, H \setminus K_v)$ its ***pseudo-external-degree***. Importantly, *pseudo-external and pseudo-anti-degrees count multiplicities of edges in the conflict graph.* We split the contribution to δ_v (Equation (4)) between external and internal neighbors:

$$\delta_v = \delta_v^e + \delta_v^a, \quad \text{where} \quad \delta_v^e \overset{\text{def}}{=} e_v - |E_v| \quad \text{and} \quad \delta_v^a \overset{\text{def}}{=} |A_v| - a_v \ . \tag{7}$$

For almost-clique K, we denote average values by $a_K = \sum_{v \in K} a_v/|K|$ and $e_K = \sum_{v \in K} e_v/|K|$.

4.3 The High-Level Algorithm

We can now describe the main steps of our algorithm. At high level, we compute the decomposition of Lemma 12, run slack generation in $V_{\mathsf{sg}} = V_{\mathsf{high}} \setminus (V_{\mathsf{cabal}} \cup V_{\mathsf{in}})$ and color each part of the decomposition in a precise order. Necessary conditions and guarantees for each step of Algorithm 1 are given in the corresponding propositions.

Parameters. Let C_1 be some large universal constant. Let us set the following parameters

$$\varepsilon = 1/2000 \ , \quad \ell = C_1 \cdot \log^{1.2} n \ , \quad \text{and} \quad r = C_1 \cdot \log^{1.1} n \ , \tag{8}$$

where ℓ is chosen to asymptotically dominate $\Theta(\log^{1.1} n)$, which is the minimum palette size for MultiColor Trial, and r sets the number of reserved colors. We call colors from $[r] = \{1, 2, \ldots, r\}$ reserved because we use them exclusively for multicolor trials. Let $V_{\mathsf{low}}, V_{\mathsf{in}}, V_\star, V_{\mathsf{dense}}$ be an ε-almost-clique decomposition of the high-degree vertices. We define

$$\mathcal{K}_{\mathsf{cabal}} = \{K : e_K < \ell\} \ , \quad V_{\mathsf{cabal}} = \{v \in V_{\mathsf{dense}} : K_v \in \mathcal{K}_{\mathsf{cabal}}\} \quad \text{and} \quad V_{\mathsf{sg}} = V_{\mathsf{high}} \setminus (V_{\mathsf{cabal}} \cup V_{\mathsf{in}}) \ .$$

Algorithm 1 The deg +1-coloring algorithm.

```
ComputeACD                                                                  (Lemma 12)
SlackGeneration in $V_{\mathsf{sg}} = V_{\mathsf{high}} \setminus (V_{\mathsf{cabal}} \cup V_{\mathsf{in}})$ without using colors $[r]$   (Lemma 11)
ColoringVstar without using colors $[r]$                                    (Proposition 13)
ColoringNonCabals                                                           (Proposition 14)
ColoringCabals                                                              (Proposition 15)
ColoringInaccurate                                                          (Proposition 16)
ColoringLowDegree                                                           (Proposition 17)
```

After running Slack Generation in V_{sg}, w.h.p., all the vertices in $V_\star$ have enough slack to get colored by MultiColor Trial. Proposition 13 achieves this coloring with additional post-conditions necessary for coloring non-cabals (Proposition 14). In words, we extend the coloring φ_{sg} produced by slack generation such that $V_\star$ is totally colored, the coloring in $V_H \setminus V_\star$ coincides with φ_{sg} and reserved colors are not used (not even in $V_\star$). Proof of Proposition 13 is given in [21, Section 4].

▶ **Proposition 13** (Coloring $V_\star$)**.** *Suppose φ_{sg} is the coloring produced by slack generation. In $\widehat{O}(\log^* n)$ rounds, we compute $\varphi \succeq \varphi_{\text{sg}}$ such that, w.h.p., we have $V_\star \subseteq \mathsf{dom}\,\varphi$, $\varphi_{|V_H \setminus V_\star} = \varphi_{\text{sg}|V_H \setminus V_\star}$ and $\varphi(V_H) \cap [r] = \emptyset$.*

Non-cabal dense vertices are colored by Algorithm 3 in [21, Section 6]. They are colored immediately after coloring $V_\star$, and the conditions needed for Proposition 14 follow from those guaranteed by Proposition 13. The algorithm combines primitives from various recent randomized coloring algorithms [42, 17, 19] (and [20]), all needing non-trivial adaptation to the current setting. Instead of applying MultiColor Trials directly after the synchronized color trial, we use the slower $O(\log\log n)$-round Slice Color algorithm of [19] to find an orientation where nodes have $O(\log n)$ uncolored out-neighbors. This allows us to use a fixed number of only $r = \Theta(\log^{1.1} n)$ reserved colors in the final application of MultiColor Trials, simplifying the (already intricate) treatment. Finally, a significant effort is needed to add up all sources of slack and show that dense vertices always have enough colors in the clique palette (see [21, Section 6.1]).

▶ **Proposition 14** (Coloring Non-Cabals)**.** *Suppose φ is a coloring such that $\mathsf{dom}\,\varphi \subseteq V_{\mathsf{sg}}$, $\varphi_{|V_{\mathsf{dense}}} = \varphi_{\text{sg}|V_{\mathsf{dense}}}$ and $\varphi(V_H) \cap [r] = \emptyset$. In $\widehat{O}(\log\log n \cdot \log^* n)$ rounds, we color all vertices in $V_{\mathsf{dense}} \setminus V_{\mathsf{cabal}}$.*

Cabals are colored by Algorithm 4 in [21, Appendix C]. The approach to color V_{cabal} is similar to Proposition 14 except for two major differences. First, vertices do not receive slack from slack generation, so we instead resort to *put-aside sets* [42]. Second, coloring put-aside sets requires a different approach that was developed in [20].

▶ **Proposition 15** (Coloring Cabals)**.** *Suppose φ is a coloring such that $V_{\mathsf{cabal}} \cap \mathsf{dom}\,\varphi = \emptyset$. Then, there exists a $\widehat{O}(\log\log n \cdot \log^* n)$-round algorithm coloring all nodes in V_{cabal} with high probability.*

The inaccurate nodes have enough slack regardless of the coloring (Equation (4)) and are easily colored at the end in the same way as $V_\star$.

▶ **Proposition 16** (Coloring Inaccurate Nodes)**.** *We can color all vertices in V_{in} in $\widehat{O}(\log^* n)$ rounds.*

Proof Sketch. The inaccuracy means that each vertex v in V_{in} has $\Omega(\varepsilon^3 \deg(v))$ colors available in $[\deg(v)+1]$ under any (possibly partial) coloring. Like for $V_\star$, we color V_{in} with $O(\varepsilon^{-12}\log\varepsilon^{-1}) = O(1)$ iterations of Random Color Trial and $O(\log^* n)$ iterations of MultiColor Trial where $\mathcal{C}(v) = [\deg(v)+1]$ and $\gamma = \Theta(\varepsilon^3)$. ◀

Low-degree nodes are colored in [21, Section 7].

▶ **Proposition 17** (Coloring Low-Degree Nodes). *Suppose φ is a coloring such that $V_{\mathsf{high}} = V_H \setminus V_{\mathsf{low}} = \mathsf{dom}\,\varphi$. In $\widehat{O}(\log^4 \log n)$ rounds, we compute a total coloring of H.*

Proof of Theorem 2. By Lemma 12, we compute the ε-almost-clique decomposition in $\widehat{O}(1/\varepsilon^6) = \widehat{O}(1)$ rounds. Running Slack Generation takes $\widehat{O}(1)$ rounds (see Algorithm 2). By Propositions 13–15, we extend the coloring to all vertices of V_{high} in $\widehat{O}(\log\log n \cdot \log^* n)$ rounds. By Proposition 17, low-degree vertices are colored in $\widehat{O}(\log^4 \log n)$ rounds. Overall, the round complexity is dominated by the coloring of low-degree vertices. ◀

5 Open Problems

The most natural immediate question following our work is:

▶ **Problem 1.** *Can we color virtual graphs in $\mathsf{cd} \cdot \mathrm{poly}(\log\log n)$ rounds using lists $\{1, 2, \dots, |N_H(v)|+1\}$ for each $v \in V_H$?*

The issue is with dense vertices whose anti-degree is hard to approximate accurately. In [20], we show that it is possible to $\Delta+1$-color in $\widehat{O}(\log^* n)$ rounds when $\Delta = \max_v |N_H(v)|$ is the maximum number of neighbors (and $\Delta \gg \log^{21} n$). However, whether the technique used to approximate anti-degrees can be generalized to $|N(v)|+1$-coloring is unclear. Using MultiColor Trials, it is possible to $(1+\varepsilon)|N_H(v)|$-color in $\mathsf{cd} \cdot \mathrm{poly}(\log\log n)$ rounds.

▶ **Problem 2.** *When is it possible to compute low-congestion support trees efficiently?*

We assumed that a support tree was given in G for each node of H (or could be easily deduced, as in the case of distance-2 coloring). It is easy, per se, to find some support tree for each node, e.g., by BFS, but this could significantly affect the congestion. It is known [30, 37, 47, 31] that for some families of graph, one can compute embeddings with low congestion. Conversely, for some problems (such as MST), on general graphs $\Omega(\sqrt{n})$ congestion is unavoidable [13]. It is a highly interesting question whether low-congestion support trees could be computed efficiently for local problems.

▶ **Problem 3.** *Can we color virtual graphs in $O((\mathsf{c}+\mathsf{d})\,\mathrm{poly}(\log\log n))$ rounds?*

Throughout the paper, our main goal was showing that coloring can be achieved in $\mathrm{poly}(\log\log n)$ rounds of broadcast and aggregation over the supports of the virtual nodes. We mostly ignored the runtime of these broadcast and aggregation operations, known to be achievable in $O(\mathsf{cd})$ rounds, and requiring $\Omega(\mathsf{c}+\mathsf{d})$. The naive runtime is already optimal in some restricted cases (when $\mathsf{c} \in O(1)$ or $\mathsf{d} \in O(1)$), but not in general. While $O(\mathsf{c}+\mathsf{d})$ schedules are known to exist for standard packet routing (with fixed paths), our problem is a proper generalization of the usual routing scenario. We also need schedules that are distributedly computable. Problem 3 asks whether our subroutines can be performed faster, possibly also pipelined, certainly an exciting open question. It is essentially an independent scheduling question, despite its implications for the main results of our paper.

▶ **Problem 4.** *Can we* $\Delta^{O(t)}$*-color* G^t *in* $O(\Delta^{\lfloor (t-1)/2 \rfloor - \Omega(1)} \operatorname{poly}\log n)$ *rounds of* CONGEST*?*

We showed that the complexity of coloring needs to grow linearly with the congestion, but this was only shown existentially for a specific class of instances. Can this dependence on congestion be avoided? In particular, the complexity of distance-3 coloring is a major open question, where congestion is necessarily linear in Δ.

References

1 Noga Alon and Sepehr Assadi. Palette sparsification beyond $(\Delta+1)$ vertex coloring. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM)*, volume 176 of *LIPIcs*, pages 6:1–6:22. LZI, 2020. doi:10.4230/LIPICS.APPROX/RANDOM.2020.6.

2 Ioannis Anagnostides, Christoph Lenzen, Bernhard Haeupler, Goran Zuzic, and Themis Gouleakis. Almost universally optimal distributed Laplacian solvers via low-congestion shortcuts. *Distributed Computing*, 36(4):475–499, 2023. doi:10.1007/S00446-023-00454-0.

3 Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear algorithms for $(\Delta+1)$ vertex coloring. In *the Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 767–786, 2019. Full version at arXiv:1807.08886. doi:10.1137/1.9781611975482.48.

4 Alkida Balliu, Sebastian Brandt, Fabian Kuhn, and Dennis Olivetti. Distributed edge coloring in time polylogarithmic in Δ. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 15–25. ACM, 2022. doi:10.1145/3519270.3538440.

5 Reuven Bar-Yehuda, Keren Censor-Hillel, Yannic Maus, Shreyas Pai, and Sriram V Pemmaraju. Distributed approximation on power graphs. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 501–510, 2020. doi:10.1145/3382734.3405750.

6 Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments.* Morgan & Claypool Publishers, 2013. doi:10.2200/S00520ED1V01Y201307DCT011.

7 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM*, 63(3):20:1–20:45, 2016. doi:10.1145/2903137.

8 Leonid Barenboim and Uri Goldenberg. Speedup of distributed algorithms for power graphs in the CONGEST model. Technical report, arXiv, 2023. doi:10.48550/arXiv.2305.04358.

9 Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The complexity of $(\Delta+1)$ coloring in congested clique, massively parallel computation, and centralized local computation. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 471–480, 2019. Full version at arXiv:1808.08419.

10 Yi-Jun Chang, Wenzheng Li, and Seth Pettie. Distributed $(\Delta+1)$-coloring via ultrafast graph shattering. *SIAM Journal of Computing*, 49(3):497–539, 2020. doi:10.1137/19M1249527.

11 Sam Coy, Artur Czumaj, Peter Davies, and Gopinath Mishra. Optimal (degree+1)-coloring in congested clique. In *the Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 261 of *LIPIcs*, pages 46:1–46:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ICALP.2023.46.

12 Artur Czumaj, Peter Davies, and Merav Parter. Simple, deterministic, constant-round coloring in congested clique and MPC. *SIAM J. Comput.*, 50(5):1603–1626, 2021. doi:10.1137/20M1366502.

13 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In *the Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 363–372, 2011. doi:10.1145/1993636.1993686.

14 Michael Elkin, Seth Pettie, and Hsin-Hao Su. $(2\Delta - 1)$-edge-coloring is much easier than maximal matching in the distributed setting. In *the Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 355–370, 2015. doi:10.1137/1.9781611973730.26.

15 Salwa Faour, Mohsen Ghaffari, Christoph Grunau, Fabian Kuhn, and Václav Rozhon. Local distributed rounding: Generalized to MIS, matching, set cover, and beyond. In *the Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 4409–4447. SIAM, 2023. doi:10.1137/1.9781611977554.CH168.

16 Manuela Fischer. Improved deterministic distributed matching via rounding. In *the Proceedings of the International Symposium on Distributed Computing (DISC)*, volume 91 of *LIPIcs*, pages 17:1–17:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.17.

17 Maxime Flin, Mohsen Ghaffari, Magnús M. Halldórsson, Fabian Kuhn, and Alexandre Nolin. Coloring fast with broadcasts. In *the Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 455–465. ACM, 2023. doi:10.1145/3558481.3591095.

18 Maxime Flin, Mohsen Ghaffari, Magnús M. Halldórsson, Fabian Kuhn, and Alexandre Nolin. A distributed palette sparsification theorem. In *the Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2024.

19 Maxime Flin, Magnús M. Halldórsson, and Alexandre Nolin. Fast coloring despite congested relays. In *the Proceedings of the International Symposium on Distributed Computing (DISC)*, 2023. doi:10.4230/LIPICS.DISC.2023.19.

20 Maxime Flin, Magnús M. Halldórsson, and Alexandre Nolin. Decentralized distributed graph coloring: Cluster graphs. Technical report, arXiv, May 2024. In submission. doi:10.48550/arXiv.2405.07725.

21 Maxime Flin, Magnús M. Halldórsson, and Alexandre Nolin. Decentralized distributed graph coloring II: degree+1-coloring virtual graphs. Technical report, arXiv, 2024. Full version of this paper. doi:10.48550/arXiv.2408.11041.

22 Sebastian Forster, Gramoz Goranci, Yang P. Liu, Richard Peng, Xiaorui Sun, and Mingquan Ye. Minor sparsifiers and the distributed laplacian paradigm. In *the Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, 2021. doi:10.1109/FOCS52979.2021.00099.

23 Pierre Fraigniaud, Magnús M. Halldórsson, and Alexandre Nolin. Distributed testing of distance-k colorings. In *the Proceedings of the International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, 2020. doi:10.1007/978-3-030-54921-3_16.

24 Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local Conflict Coloring. In *the Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 625–634, 2016. doi:10.1109/FOCS.2016.73.

25 Marc Fuchs and Fabian Kuhn. List defective colorings: Distributed algorithms and applications. In *the Proceedings of the International Symposium on Distributed Computing (DISC)*, LIPIcs, 2023. doi:10.4230/LIPICS.DISC.2023.22.

26 Mohsen Ghaffari. Near-optimal scheduling of distributed algorithms. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 3–12, 2015. doi:10.1145/2767386.2767417.

27 Mohsen Ghaffari and Christoph Grunau. Faster deterministic distributed MIS and approximate matching. In *the Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 1777–1790. ACM, 2023. doi:10.1145/3564246.3585243.

28 Mohsen Ghaffari, Christoph Grunau, Bernhard Haeupler, Saeed Ilchi, and Václav Rozhon. Improved distributed network decomposition, hitting sets, and spanners, via derandomization. In *the Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2532–2566. SIAM, 2023. doi:10.1137/1.9781611977554.CH97.

29 Mohsen Ghaffari, Christoph Grunau, and Václav Rozhoň. Improved deterministic network decomposition. In *the Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2021. arXiv:2007.08253.

30 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: Low-congestion shortcuts, MST, and Min-Cut. In *the Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 202–219. SIAM, 2016. doi:10.1137/1.9781611974331.CH16.

31 Mohsen Ghaffari and Bernhard Haeupler. Low-congestion shortcuts for graphs excluding dense minors. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 213–221. ACM, 2021. doi:10.1145/3465084.3467935.

32 Mohsen Ghaffari, Bernhard Haeupler, and Goran Zuzic. Hop-constrained oblivious routing. In *the Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 1208–1220. ACM, 2021. doi:10.1145/3406325.3451098.

33 Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. Near-optimal distributed maximum flow. *SIAM J. Comput.*, 47(6):2078–2117, 2018. doi:10.1137/17M113277X.

34 Mohsen Ghaffari and Fabian Kuhn. Distributed minimum cut approximation. In *International Symposium on Distributed Computing*, pages 1–15. Springer, 2013. doi:10.1007/978-3-642-41527-2_1.

35 Mohsen Ghaffari and Fabian Kuhn. Deterministic distributed vertex coloring: Simpler, faster, and without network decomposition. In *the Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 1009–1020. IEEE, 2021. doi:10.1109/FOCS52979.2021.00101.

36 Mohsen Ghaffari and Goran Zuzic. Universally-optimal distributed exact min-cut. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 281–291. ACM, 2022. doi:10.1145/3519270.3538429.

37 Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Low-congestion shortcuts without embedding. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 451–460. ACM, 2016. doi:10.1145/2933057.2933112.

38 Bernhard Haeupler, Shyamal Patel, Antti Roeyskoe, Cliff Stein, and Goran Zuzic. Polylog-competitive deterministic local routing and scheduling. *CoRR*, abs/2403.07410, 2024. doi:10.48550/arXiv.2403.07410.

39 Magnús M. Halldórsson, Fabian Kuhn, and Yannic Maus. Distance-2 coloring in the CONGEST model. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 233–242, 2020. doi:10.1145/3382734.3405706.

40 Magnús M. Halldórsson, Fabian Kuhn, Yannic Maus, and Alexandre Nolin. Coloring fast without learning your neighbors' colors. In *the Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 39:1–39:17, 2020. doi:10.4230/LIPIcs.DISC.2020.39.

41 Magnús M. Halldórsson, Fabian Kuhn, Yannic Maus, and Tigran Tonoyan. Efficient randomized distributed coloring in CONGEST. In *the Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 1180–1193. ACM, 2021. Full version at arXiv:2105.04700.

42 Magnús M. Halldórsson, Fabian Kuhn, Alexandre Nolin, and Tigran Tonoyan. Near-optimal distributed degree+1 coloring. In *the Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 450–463. ACM, 2022. doi:10.1145/3519935.3520023.

43 Magnús M. Halldórsson and Alexandre Nolin. Superfast coloring in CONGEST via efficient color sampling. *Theor. Comput. Sci.*, 948:113711, 2023. doi:10.1016/J.TCS.2023.113711.

44 Magnús M. Halldórsson, Alexandre Nolin, and Tigran Tonoyan. Overcoming congestion in distributed coloring. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 26–36. ACM, 2022. doi:10.1145/3519270.3538438.

45 David G. Harris, Johannes Schneider, and Hsin-Hao Su. Distributed $(\Delta + 1)$-coloring in sublogarithmic rounds. *Journal of the ACM*, 65:19:1–19:21, 2018. doi:10.1145/3178120.

46 D. Hefetz, Y. Maus, F. Kuhn, and A. Steger. A polynomial lower bound for distributed graph coloring in a weak LOCAL model. In *the Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 99–113, 2016.

47 Shimon Kogan and Merav Parter. Low-congestion shortcuts in constant diameter graphs. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 203–211. ACM, 2021. doi:10.1145/3465084.3467927.

48 Frank Thomson Leighton, Bruce M. Maggs, Abhiram G. Ranade, and Satish Rao. Randomized routing and sorting on fixed-connection networks. *J. Algorithms*, 17(1):157–205, 1994. doi:10.1006/JAGM.1994.1030.

49 Frank Thomson Leighton, Bruce M. Maggs, and Satish Rao. Packet routing and job-shop scheduling in O(congestion + dilation) steps. *Combinatorica*, 14(2):167–186, 1994. doi:10.1007/BF01215349.

50 Frank Thomson Leighton, Bruce M. Maggs, and Andréa W. Richa. Fast algorithms for finding O(congestion + dilation) packet routing schedules. *Combinatorica*, 19(3):375–401, 1999. doi:10.1007/S004930050061.

51 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 42–50. ACM, 2013. doi:10.1145/2484239.2501983.

52 Nati Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

53 Zvi Lotker, Boaz Patt-Shamir, and David Peleg. Distributed MST for constant diameter graphs. *Distributed Computing*, 18(6):453–460, 2006. doi:10.1007/S00446-005-0127-6.

54 M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15:1036–1053, 1986. doi:10.1137/0215074.

55 Yannic Maus, Saku Peltonen, and Jara Uitto. Distributed symmetry breaking on power graphs via sparsification. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 157–167. ACM, 2023. doi:10.1145/3583668.3594579.

56 Yannic Maus and Tigran Tonoyan. Local conflict coloring revisited: Linial for lists. In *the Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 16:1–16:18, 2020. doi:10.4230/LIPIcs.DISC.2020.16.

57 Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM J. on Comp.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

58 Bruce A. Reed. ω, Δ, and χ. *J. Graph Theory*, 27(4):177–212, 1998. doi:10.1002/(SICI)1097-0118(199804)27:4<177::AID-JGT1>3.0.CO;2-K.

59 Václav Rozhon, Christoph Grunau, Bernhard Haeupler, Goran Zuzic, and Jason Li. Undirected $(1+\varepsilon)$-shortest paths via minor-aggregates: near-optimal deterministic parallel and distributed algorithms. In *the Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 478–487, 2022. doi:10.1145/3519935.3520074.

60 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *the Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 350–363, 2020.

61 Johannes Schneider and Roger Wattenhofer. A new technique for distributed symmetry breaking. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 257–266. ACM, 2010. doi:10.1145/1835698.1835760.

A Color Trials

We state here some classical symmetry breaking algorithms with additional assumptions for virtual graphs. The main difference with LOCAL or CONGEST version of these algorithm is that vertices sample in color space $\mathcal{C}(v)$ instead of their palette.

▶ **Lemma 18** (Random Color Trial). *Let $\gamma \in (0,1)$ be universal constants known to all nodes. Let φ be a coloring, $S \subseteq V \setminus \operatorname{dom}\varphi$ a set of uncolored nodes, and sets $\mathcal{C}(v) \subseteq [\deg(v)+1]$ for each $v \in S$ such that*

1. *v can sample a uniform color in $\mathcal{C}(v)$ in $O(1)$ rounds,*
2. *$|\mathcal{C}(v)| \geqslant \Theta(\gamma^{-1}\log n)$,*
3. *$|L_\varphi(v) \cap \mathcal{C}(v)| \geqslant \gamma|\mathcal{C}(v)|$, and*
4. *$|L_\varphi(v) \cap \mathcal{C}(v)| \geqslant \gamma|N_\varphi(v) \cap S|$.*

Let $\psi \succeq \varphi$ be the coloring produced by `TryColor`*. Then, w.h.p., each $w \in V_H$ has uncolored degree in S*

$$|N_\psi(w) \cap S| \leqslant \max\left\{(1-\gamma^4/64)|N_\varphi(w) \cap S|,\ \Theta(\gamma^{-4}\log n)\right\} .$$

The algorithm ends after $\widehat{O}(1)$ rounds and $\psi(v) \in \mathcal{C}(v)$ for all $v \notin \operatorname{dom}\varphi$.

The MultiColorTrial in Lemma 19 is adapted from [43] to sample colors from a restricted known color space.

▶ **Lemma 19** (MultiColorTrial, adapted from [43]). *Let φ be a (partial) coloring of H, $S \subseteq V_H \setminus \operatorname{dom}\varphi$, and $\mathcal{C}(v) \subseteq [\deg(v)+1]$ be a color space for each node. Suppose that there exists some constant $\gamma > 0$ known to all nodes such that*

1. *$\mathcal{C}(v)$ is known to all machines in $V(v)$; and*
2. *$|L_\varphi(v) \cap \mathcal{C}(v)| - |N_\varphi(v) \cap S| \geqslant \max\{2|N_\varphi(v) \cap S|, \Theta(\log^{1.1} n)\} + \gamma|\mathcal{C}(v)|$.*

Then, there exists an algorithm computing a coloring $\psi \succeq \varphi$ such that, w.h.p., all nodes of S are colored and $\psi(v) \in \mathcal{C}(v)$ for each $v \in S$. The algorithm runs in $\widehat{O}(\gamma^{-1}\log^* n)$ rounds.

B Pseudo-Code

Algorithm 2 `SlackGeneration`.

```
Each $v \in V_{\mathsf{sg}}$ joins $V^{\mathsf{active}}$ w.p. $p_{\mathsf{g}} = 1/20$.
Each $v \in V^{\mathsf{active}}$ samples $c(v) \in \{r+1, r+2, \dots, \deg(v)+1\}$ uniformly at random.
Let $\varphi_{\mathrm{sg}}(v) = c(v)$ if $v \in V^{\mathsf{active}}$ and $c(v) \notin c(N_H^+(v))$. Otherwise, set $\varphi_{\mathrm{sg}}(v) = \bot$.
```

Algorithm 3 `ColoringNonCabals`.

```
Input: A coloring $\varphi$ such as described in Proposition 14
Output: A coloring $\psi \succeq \varphi$ such that $V_{\mathsf{dense}} \setminus V_{\mathsf{cabal}} = \operatorname{dom}\psi$
ColorfulMatching when $a_K \geqslant \Omega(\log n)$  // Let $\varphi_{\mathrm{cm}}$ be the coloring produced
ColoringOutliers with $\mathcal{C}(v) = [r+1, \deg(v)+1]$
SynchronizedColorTrial
TryColor for $O(1)$ rounds with $\mathcal{C}(v) = L_\varphi(K_v) \cap [r+1, \deg(v)+1]$
SliceColor with $\mathcal{C}(v) = L_\varphi(K_v) \cap [r+1, \deg(v)+1]$
Let $\mathcal{L}_1, \dots, \mathcal{L}_{O(\log\log n)}$ be the layers produced by SliceColor.
for $i = O(\log\log n)$ to 1 do
    MultiColorTrial with $\mathcal{C}(v) = [r]$ in $\mathcal{L}_i$
```

Algorithm 4 Cabals.

```
Let $r' \overset{\text{def}}{=} 150\ell$, where $\ell = C_1 \log^{1.2} n$ is as described in Equation (8).
ColorfulMatching.
ColoringOutliers with $\mathcal{C}(v) = [\deg(v)+1] \setminus [r']$.
ComputePutAside $P_K \subseteq I_K$.
SynchronizedColorTrial with $S_K = K \setminus (\mathsf{dom}\,\varphi \cup P_K)$
SliceColor with $\mathcal{C}(v) = [\deg(v)+1] \setminus [r']$
Let $\mathcal{L}_1, \dots, \mathcal{L}_{O(\log\log n)}$ be the layers produced by SliceColor
for $i = O(\log\log n)$ to 1 do
    MultiColorTrial with $\mathcal{C}(v) = [r']$ in $\mathcal{L}_i$
ColorPutAsideSets
```

Distributed Model Checking on Graphs of Bounded Treedepth

Fedor V. Fomin
University of Bergen, Norway

Pierre Fraigniaud
IRIF, Université Paris Cité and CNRS, Paris, France

Pedro Montealegre
Universidad Adolfo Ibañez, Santiago, Chile

Ivan Rapaport
Universidad de Chile, Santiago, Chile

Ioan Todinca
LIFO, Université d'Orléans, France
INSA Centre-Val de Loire, Orléans, France

Abstract

We establish that every monadic second-order logic (MSO) formula on graphs with bounded treedepth is decidable in a constant number of rounds within the CONGEST model. To our knowledge, this marks the first meta-theorem regarding distributed model-checking. Various optimization problems on graphs are expressible in MSO. Examples include determining whether a graph G has a clique of size k, whether it admits a coloring with k colors, whether it contains a graph H as a subgraph or minor, or whether terminal vertices in G could be connected via vertex-disjoint paths. Our meta-theorem significantly enhances the work of Bousquet et al. [PODC 2022], which was focused on *distributed certification* of MSO on graphs with bounded treedepth. Moreover, our results can be extended to solving optimization and counting problems expressible in MSO, in graphs of bounded treedepth.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Verification by model checking

Keywords and phrases proof-labeling schemes, local computing, CONGEST model

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.25

Related Version *Full Version*: https://arxiv.org/abs/2405.03321 [28]

Funding *Fedor V. Fomin*: the research leading to these results has received funding from the Research Council of Norway via the project BWCA (grant no. 314528).
Pierre Fraigniaud: partially supported by the ANR projects DUCAT (ANR-20-CE48-0006), and QuData (ANR-18-CE47-0010).
Pedro Montealegre: P.M. has received funding from FONDECYT 1230599.
Ivan Rapaport: I.R. has received funding from FB210005, BASAL funds for centers of excellence from ANID- Chile, and FONDECYT 1220142.

1 Introduction

Distributed *decision* [12, 30, 31] and distributed *certification* [32, 43, 51] are two complementary fields of distributed computing, closely associated with distributed fault-tolerant computing [3, 19, 50]. Both fields are addressing the problem of checking whether a distributed system is in a legal state with respect to a given specification, or not. We examine this problem in the classical context of distributed computing in networks, under the standard

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 25; pp. 25:1–25:20
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

CONGEST model [60]. Recall that this model assumes networks modeled as simple connected n-node graphs, in which every node is provided with an identifier on $O(\log n)$ bits that is unique in the network. Computation proceeds synchronously as a sequence of *rounds*. At each round, every node sends a message to each of its neighbors in the graph, receives the messages sent by its neighbors, and performs some individual computation. A crucial point is that messages are restricted to be of size $O(\log n)$ bits. While this suffices to transmit an identifier, or a constant number of identifiers, transmitting large messages may require multiple rounds, typically $\Theta(k/\log n)$ rounds for k-bit messages.

Distributed Decision

Given a boolean predicate Π on graphs, e.g., whether the graph is H-free for some fixed graph H, a *decision* algorithm for Π takes as input a graph $G = (V, E)$, and outputs whether G satisfies Π or not. Specifically, every node v receives as input its identifier $\mathsf{id}(v)$, and, after a certain number of rounds of communication with its neighbors, it outputs *accept* or *reject*, under the constraint that G satisfies Π if and only if the output of each of the nodes $v \in V$ is accept. In other words,

$$G \models \Pi \iff \forall v \in V(G),\ \text{output}(v) = \text{accept}.$$

Some predicates are easy to decide *locally*, i.e., in a constant number of rounds. A canonical example is checking whether the (connected) graph G is regular, for which one round suffices. However, other predicates cannot be checked locally, with canonical example checking whether there is a unique node of degree 3 in the network. Indeed, checking this property requires $\Omega(n)$ rounds in networks of diameter $\Theta(n)$, as two nodes of degree 3 may be at arbitrarily large distances in the graph. Another example of a difficult problem is checking whether the graph is C_4-free, i.e., does not contain a 4-cycle as a subgraph, which requires $\tilde{\Omega}(\sqrt{n})$ rounds [13]. One way to circumvent the difficulty of local checkability, i.e., to address graph predicates requiring a large number of rounds for being decided, is to consider distributed *certification*.

Distributed Certification

A *certification scheme* for a boolean predicate Π is a pair *prover-verifier* (see [19] for more details). The prover is a centralized, computationally unbounded, non-trustable oracle. Given a graph $G = (V, E)$, the prover assigns a *certificate* $c(v) \in \{0, 1\}^\star$ to each node $v \in V$. These certificates are forged by the prover using the complete knowledge of the graph G. The verifier is a distributed 1-round algorithm. Each node v takes as sole input its identifier $\mathsf{id}(v)$ and its certificate $c(v)$. In particular, for distributed decisions, v is unaware of the graph G. Every node v just exchanges once its identifier and certificate with its neighbors, and then it must output *accept* or *reject*.

The certification scheme is correct if the following two conditions hold. The *completeness* condition states that if G satisfies Π, then the oracle can provide the nodes with certificates that they all accept. The *soundness* condition says that if G does not satisfy Π, then no matter the certificates assigned by the oracle to the nodes, at least one of them rejects. That is, the role of the verifier is to check that the collection of certificates assigned to the nodes by the prover is indeed a global proof that the graph satisfies the predicate. In other words,

$$G \models \Pi \iff \exists c : V(G) \to \{0, 1\}^\star \ :\ \forall v \in V(G),\ \text{output}(v) = \text{accept}.$$

The main measure of complexity of a certification scheme is the maximum *size* of the certificates assigned by the prover to the nodes on legal instances, i.e., for graphs G satisfying the given predicate. Ideally, to be implemented in a single round under the CONGEST model, the certificates should be of size $O(\log n)$ bits. Interestingly, many graph properties can be certified with such short certificates, including acyclicity [51], planarity [25], bounded genus [17, 26], etc. On the other hand, basic graph properties require large certificates, including diameter 2 vs. 3 (requiring $\tilde{\Omega}(n)$-bit certificates [8]), non-3-colorability (requiring $\tilde{\Omega}(n^2)$-bit certificates [43]), C_4-freeness (requiring $\tilde{\Omega}(\sqrt{n})$-bit certificates [13]), etc. The following question was thus raised, under different formulations (see, e.g., [20]): *What are the graph properties that admit certification schemes with $O(\log n)$-bit certificates, or, to the least, certificates of polylogarithmic size?* Answering this question requires formalizing the notion of "graph predicate".

Monadic Second-Order Logic

Recall that, in the *first-order* logic (FO) of graphs, a graph property is expressed as a quantified logical sentence whose variables represent vertices, with predicates for equality $(=)$ and adjacency (adj). An FO formula is therefore constructed according to the following set of rules, where x and y are vertices, and φ and ψ are FO formulas:

$$x = y \mid \mathsf{adj}(x,y) \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \neg\varphi \mid \exists x\varphi \mid \forall x\varphi.$$

As an example, triangle-freeness can be formulated as

$$\varphi = \neg\exists x_1\exists x_2\exists x_3\big(\mathsf{adj}(x_1,x_2) \wedge \mathsf{adj}(x_2,x_3) \wedge \mathsf{adj}(x_3,x_1)\big).$$

The formula above assumes simple graphs (i.e., no loops nor multiple edges). If the graphs may have loops, then one should add the predicate $\neg(x_i = x_j)$ to the formula for every $i \neq j$.

The *monadic second-order* logic (MSO) extends FO by allowing quantification on *sets* of vertices and edges, with the incidence predicate $\mathsf{inc}(v,e)$ indicating whether vertex v is incident to edge e, and the membership $(\in)$ predicate. For instance, acyclicity can be formulated as

$$\varphi = \neg\exists X \neq \varnothing\ \forall x \in X\ \exists y_1 \in X\ \exists y_2 \in X\big(\neg(y_1 = y_2) \wedge \mathsf{adj}(x,y_1) \wedge \mathsf{adj}(x,y_2)\big).$$

Note that $X \neq \varnothing$ can merely be written as $\exists x \in X$. Note also that acyclicity cannot be expressed in FO as the length of the potential cycle is unbounded, from which it follows that one cannot quantify on vertices only for expressing acyclicity, because one does not know *how many* vertices should be considered. On the other hand, since FO can express properties such as C_4-freeness, which, as mentioned before, requires certificates on $\tilde{\Omega}(\sqrt{n})$ bits, there is no hope of establishing a *meta-theorem* about FO regarding compact certification in all graphs. Nevertheless, a breakthrough in the theory of distributed certification was recently obtained by Bousquet, Feuilloley, and Pierron [20], who showed that every MSO predicate admits a distributed certification scheme with $O(\log n)$-bit certificates in the family of graphs with bounded *treedepth*.

Algorithmic Meta-Theorems

A vibrant line of research in sequential computing is the development of algorithmic meta-theorems. According to Grohe and Kreutzer [45], algorithmic meta-theorems assert that certain families of algorithmic problems, typically defined by some logical and combinatorial conditions, can be solved efficiently under some suitable definition of this term. Such theorems play an essential role in the theory of algorithms as they reveal a profound interplay between algorithms, logic, and combinatorics. One of the most celebrated examples of a meta-theorem is Courcelle's theorem, which asserts that graph properties definable in MSO are decidable in linear time on graphs of bounded treewidth [10]. For an introduction to this fascinating research area, we refer to the surveys by Kreutzer [52] and Grohe [44].

Bousquet, Feuilloley, and Pierron in [20] introduced the exploration of algorithmic meta-theorems in distributed computing. Their primary result in this direction is that any MSO formula can be locally *certified* on graphs with bounded treedepth using a logarithmic number of bits per node, which represents the golden standard in certification. This theorem has numerous consequences for certification – for more details, we refer to [20]. Notably, the FO property C_4-freeness, and the MSO property non-3-colorability, which both necessitate certificates of polynomial size in general, can be certified with just $O(\log n)$-bit certificates in graphs of bounded treedepth. Bousquet et al.'s result has been extended to more comprehensive classes of graphs, including graphs excluding a small minor [6], as well as graphs of bounded *treewidth*, and graphs of bounded *cliquewidth*. However, this extension comes at the cost of slightly larger certificates, of $O(\log^2 n)$ bits, as seen in [34] and [33], respectively.

With significant advances in developing meta-theorems for distributed *certification*, there's a notable absence of similar results for distributed *decision*. It prompts a natural question: could such results be obtained for the round-complexity of CONGEST? More concretely, the fundamental inquiry that remains unaddressed by Bousquet et al.'s paper, and by the subsequent works regarding distributed certification of MSO predicates is:

Question. *What is the round-complexity in* CONGEST *of deciding MSO formulas in graphs of bounded treedepth?*

A first step in answering this question was proposed in [59] where it is stated that, in any graph class of treedepth at most d, for every fixed connected graph H, H-freeness can be decided in $O(1)$ rounds in CONGEST. In this paper, we offer a comprehensive answer to the question. To elucidate our results, we first need to define the treedepth of a graph.

Treedepth

For any non-negative integer d, a (connected) graph G has treedepth at most d if there exists a rooted tree T spanning the vertices of G, with depth at most d, such that, for every edge $\{u, v\}$ in G, u is an ancestor of v in T, or v is an ancestor of u in T, see Fig. 1. The treedepth of a graph G, denoted by $\mathtt{td}(G)$, is the smallest d for which such a tree exists.

The class of graphs with bounded treedepth, i.e., of treedepth d for some fixed $d \geq 0$, has strong connections with minor-closed families of graphs. Specifically, for any family $\mathcal{F}$ of graphs closed under taking graph minors, the graphs in $\mathcal{F}$ have bounded treedepth if and only if $\mathcal{F}$ does not include all the paths [58]. Similarly, the graphs with bounded treedepth have a finite set of forbidden induced subgraphs, and any property of graphs monotonic with respect to induced subgraphs can be tested in polynomial time on graphs of bounded treedepth [58]. Computing the treedepth of a graph is NP-hard, but since treedepth is monotonic under graph minors, it is fixed-parameter tractable (FPT) [36]. Last but not least, MSO and FO have the same expressive power in graph classes of bounded treedepth [14].

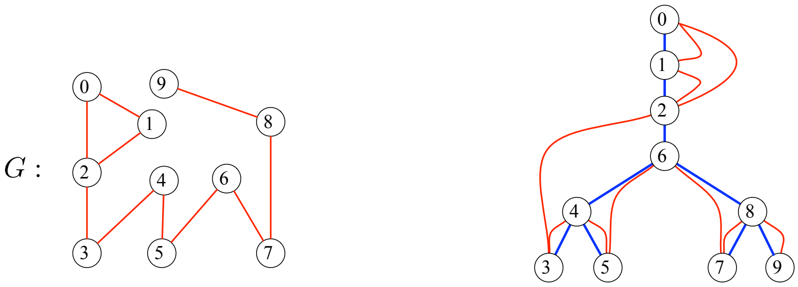


Figure 1 Embedding of a graph G into a tree T of depth 6.

1.1 Our Results

We prove that, for every MSO formula φ, there is an algorithm $\mathcal{A}$ that, for every n-node graph G, decides whether $G \models \varphi$ in $O(2^{2^{\mathsf{td}(G)}})$ rounds in the CONGEST model. That is, the round-complexity of $\mathcal{A}$ depends only on the treedepth of the input graph, and on the MSO formula, i.e., it does not depend on the *size* n of the graph. Thus $\mathcal{A}$ performs a constant number of rounds in any class of graphs with bounded treedepth. In particular, deciding non-3-colorability can be done in $O(1)$ rounds in graphs of bounded treedepth, in contrast to general graphs, for which deciding non-3-colorability requires a polynomial number of rounds by [43].

Our meta-theorem is essentially the best that one may hope regarding distributed model checking MSO formulas in a constant number of rounds in CONGEST. Indeed, the FO predicate "there is at least one vertex of degree > 2" requires $\Omega(n)$ rounds to be checked in this class. Hence our theorem cannot be extended to graphs of bounded treewidth or bounded cliquewidth, actually not even to bounded pathwidth, and not even to the class $\mathcal{P} \cup \mathcal{B}$ where $\mathcal{P}$ is the set of all paths, and $\mathcal{B}$ is the set of all graphs composed of a path to which is attached a claw at one of its endpoints.

We also consider distributed model checking of *labeled* graphs. For instance, one can check whether a given set of vertices is a feedback vertex set, i.e., whether the graph obtained by removing this set of vertices is acyclic. For such a predicate, it is sufficient to add a unary predicate to the logical structure used to mark the nodes, say $\mathsf{mark}(x) = \mathit{true}$ means that vertex x is in the set. Using this unary predicate, φ can express the fact that there are no cycles in G passing only trough nodes x for which $\mathsf{mark}(x) = \mathit{false}$. As another example, the fact that a graph is properly 2-colored can be expressed using two unary boolean predicates red and blue, as

$$\varphi = \Big(\forall x\ \big(\mathsf{red}(x) \vee \mathsf{blue}(x)\big)\Big) \wedge \Bigg(\forall x, y\ \neg\Big(\mathsf{adj}(x,y) \wedge \big((\mathsf{red}(x) \wedge \mathsf{red}(y)) \vee (\mathsf{blue}(x) \wedge \mathsf{blue}(y))\big)\Big)\Bigg).$$

Since we also deal with MSO, we can also label edges. For instance, one can check whether a given set of edges forms a spanning tree. Indeed, it is sufficient to introduce a unary predicate used to mark the edges: $\mathsf{mark}(e) = \mathit{true}$ means that edge e is in the set. As for feedback vertex set, using this unary predicate, φ can express the fact that the set of marked edges is a spanning tree (i.e., every node is incident to at least one marked edge, and any two vertices are connected by a path of marked edges). We show that deciding MSO formulas on *labeled* graphs of bounded treedepth can be done in $O(1)$ rounds in the CONGEST model.

More generally, we also consider the *optimization* variants of decision problems expressible in MSO on graphs of bounded treedepth. For instance, an independent set can be expressed as an MSO formula with a free variable S, such as $\varphi(S) = \forall x \in S\ \forall y \in S\ \neg\mathsf{adj}(x,y)$. Then,

$max\varphi$, i.e., maximum independent set, consists in, given any graph $G = (V, E)$, finding the largest set $S \subseteq V$ such that $G \models \varphi(S)$. We show that, for every MSO formula $\varphi(S)$ with free variable $S \subseteq V$ or $S \subseteq E$, there is an algorithm for graphs of bounded treedepth solving $max\varphi$ (and $min\varphi$) in a constant number of rounds in the CONGEST model. This constant is of the form $O(g(\mathsf{td}(G), \varphi))$ for some function g. Thanks to the expressive power of MSO, our results yield algorithms with a constant number of rounds in the CONGEST model on graphs of bounded treedepth for numerous popular optimization problems including minimum vertex cover, minimum feedback vertex set, minimum dominating set, maximum independent set, maximum induced forest, maximum clique, maximum matching, minimum spanning tree, Hamiltonian cycle, cubic subgraph, planar subgraph, Eulerian subgraph, Steiner tree, disjoint paths, min-cut, minor and topological minor containment, rural postman, k-colorability, edge k-colorability, partition into k cliques, and covering by k cliques. We also extend our results to *counting* problems, such as counting triangles or perfect matchings. Due to space restriction we only present in details in this article the distributed model checking result, for unlabeled graphs. For the optimization and counting variants, and the general case of labeled graphs, we refer to the full paper [28].

Finally, we briefly discuss some applications of our results to much larger classes of graphs, namely graphs of bounded *expansion* (see [58] for an extended introduction). Graphs of bounded expansion include planar graphs, and more generally, all classes of graphs defined from excluding minor. It was shown [59] that, for every class $\mathcal{G}$ of graphs with bounded expansion, and every positive integer p, there is an algorithms performing in $O(\log n)$ rounds under the CONGEST model that partitions the vertex set V of any graph $G = (V, E) \in \mathcal{G}$ into $f(p)$ parts $V_1, \ldots, V_{f(p)}$ such that every collection $V_{i_1}, \ldots, V_{i_q}$ of at most p parts, $1 \leq q \leq p$, $\{i_1, \ldots, i_q\} \subseteq \{1, \ldots, f(p)\}$, induces a (not necessarily connected) subgraph of G with treedepth at most p. The function f solely depends on the considered class $\mathcal{G}$ of bounded expansion. The vertex partitioning $V_1, \ldots, V_{f(p)}$ is called a low treedepth decomposition with parameter p. Plugging in our techniques into this framework, we show that, for every connected graph H, H-freeness can be decided in $O(\log n)$ rounds under the CONGEST model in any class of graphs with bounded expansion. This result was claimed in [59] with no proofs. We provide that claim with a complete formal proof.

1.2 Other Related Work

The quest for efficient (sublinear) algorithms for solving classical graph problems in the CONGEST model dates back to the seminal paper by Garay, Kutten and Peleg [38], where an algorithm for constructing an MST was designed. Since then, a long series of problems have been addressed, such as connectivity decomposition [7], tree embeddings [41] k-dominating set [53], stiener trees [54], min-cut [40, 56], max-flow [39], shortest path [47, 55], among others. Additionally, algorithms tailored to specific classes of networks have also been developed: DFS for planar graphs [42], MST for bounded genus graphs [46], MIS for networks excluding a fixed minor [9], etc.

Distributed certification is a very vivid topic, and many results have appeared since the survey [21]. A handful of recent papers considered *approximate* variants of the problem, a la property testing [15, 16, 22]. In particular, it was shown that every monotone (i.e., closed under taking subgraphs) and summable (i.e., stable by disjoint union) property Π has a compact approximate certification scheme in any proper minor-closed graph class [18]. Other recent contributions dealt with augmenting the power of the verifier in certification schemes, which includes tradeoffs between the size of the certificates and the number of rounds of the verification protocol [24], randomized verifiers [35], quantum verifiers [29], and several hierarchies of certification mechanisms, including games between a prover and a disprover [1, 23], interactive protocols [11, 49, 57], and even zero-knowledge distributed certification [2], and distributed quantum interactive protocols [37].

2 Treedepth and treewidth

Throughout the paper, trees (or forests) are considered as rooted. The *depth* of a tree is the number of vertices of a longest path from the root to a leaf. The depth of a forest is the maximum depth among its trees. Let us recall the definition of the treedepth. The interested reader can refer to the book of Nešetřil and Ossona de Mendez [58] for further insights.

▶ **Definition 1** (treedepth). *The* treedepth *of a graph $G = (V, E)$ is the minimum depth of a forest $T = (V, F)$ on the same vertex set as G, such that, for any edge $\{u, v\}$ of G, one of the endpoints is an ancestor of the other in the forest T. Such a forest T is also called* elimination forest *of G.*

Observe that if G is connected then the forest T in the definition above is actually a tree. Also, Definition 1 does not require the forest T to be a subgraph of G. The following statement is an alternative, equivalent definition for treedepth. This recursive definition implicitly provides a recursive construction of an elimination tree.

▶ **Lemma 2** ([58]). *The treedepth of a graph G is:*

$$\mathsf{td}(G) = \begin{cases} 1 & \textit{if } G \textit{ has a unique vertex,} \\ 1 + \min_{v \in V(G)} \mathsf{td}(G - v) & \textit{if } G \textit{ is connected,} \\ \max\{\mathsf{td}(C) \mid C \textit{ is a connected component of } G\} & \textit{otherwise.} \end{cases}$$

On the other hand, tree decompositions and treewidth of graphs were introduced by Robertson and Seymour [61].

▶ **Definition 3** (treewidth). *A* tree decomposition *of a graph $G = (V, E)$ is a pair (T, B) where $T = (I, F)$ is a tree, and $B = \{B_i, i \in I\}$ is a collection of subsets of vertices of G, called* bags*, such that the following conditions hold:*

- *For every vertex of G, there exists some bag containing this vertex;*
- *For every edge e of G there is some bag containing both endpoints of e;*
- *For every $v \in V$, the set $\{i \in I : v \in B_i\}$ of bags containing v forms a connected subgraph of T.*

The width *of a tree decomposition is the maximum size of a bag, minus one. The* treewidth *of a graph G, denoted by $\mathsf{tw}(G)$, is the smallest width of a tree decomposition of G.*

It is known [58] that the treedepth of a graph is at least its treewidth. Given an elimination tree T of a graph G, we can define a canonical tree decomposition associated to this same tree, such that the width of the decomposition corresponds to the depth of T, minus one. The following lemma is a straightforward consequence of the definitions of elimination trees and of tree decompositions.

▶ **Lemma 4** (canonical tree decomposition). *Let $T = (V, F)$ be an elimination tree of depth d of a graph $G = (V, E)$. Let us associate to each node u of T a bag $B(u)$ containing u and all the ancestors of u in T. Then $T = (V, F)$, and the corresponding set of bags $(B_u)_{u \in V}$, form a tree-decomposition of G, of width $d - 1$.*

For instance, the treedepth of an n-vertex path P_n is exactly $\lceil \log(n+1) \rceil$ (see, e.g., [58]). The treedepth of a graph does not increase when we delete some of its edges or vertices. Thus graphs of treedepth d have no paths on 2^d vertices. This observation yields the following lemma.

▶ **Lemma 5.** *Let $T = (V, F)$ be an elimination tree of a graph $G = (V, E)$ with $F \subseteq E$. Then the depth of T is at most $2^{\mathsf{td}(G)}$.*

Proof. Let $d = \mathsf{td}(G)$. Assume, for the purpose of contradiction, that T has depth larger than 2^d. It follows that the longest path P in T from its root to a leaf contains at least 2^d vertices. The path P is also a path in G, so the treedepth of P is at most the treedepth of G, i.e., at most d. This is a contradiction with the fact that, for n-node paths, $\mathsf{td}(P_n) = \lceil \log(n+1) \rceil$. ◀

3 Tree decompositions and w-terminal recursive graphs

Courcelle's theorem [10] states that any property expressible in MSO can be decided in linear (sequential) time on graphs of bounded treewidth. We use an alternative proof of Courcelle's theorem, by Borie, Parker, and Tovey [5]. Indeed, this proof provides us with an explicit dynamic programming strategy, which will be used in our distributed protocol.

Graphs of bounded treewidth can also be defined recursively, based on a graph grammar. Let w be a positive integer. A *w-terminal graph* is a triple (V, W, E) where $G = (V, E)$ is a graph, and $W \subseteq V$ is a *totally ordered* set of at most w distinguished vertices. Vertices of W are called the *terminals* of the graph, and we denote by $\tau(G)$ the number of its terminals. As the terminal set W is totally ordered, we can refer to the rth terminal, for $1 \leq r \leq w$. Moreover, since vertices are given distinct identifiers in CONGEST, one can view W as ordered by these identifiers.

The class of w-terminal *recursive* graphs is defined, starting from w-terminal *base* graphs, by a sequence of *compositions*, or *gluings*. A w-terminal base graph is a w-terminal graph of the form (V, W, E) with $W = V$. A composition f acts on two[1] w-terminal graphs, and produces a new w-terminal graph, as follows (see Figure 2 for an example[2], for $w = 2$).

The graph $G = f(G_1, G_2)$ is obtained by, first, making disjoint copies of the two graphs G_1 and G_2, and, second, "gluing" together some terminals of G_1 and G_2. In the gluing operation, each terminal of G_1 is identified with at most one terminal of G_2. Formally, the gluing performed by f is represented by a matrix $m(f)$ having $\tau(G) \leq w$ rows, and two columns, with integer entries in $\{0, \dots, \tau(G)\}$. At row r of the matrix, $m_{r,c}(f)$ indicates which terminal of each graph G_c, $c \in \{1, 2\}$ is identified to the rth terminal of graph G. If $m_{r,c}(f) = 0$, then no terminal of G_c is identified to terminal r of G (in particular, if $m_{r,1}(f) = m_{r,2}(f) = 0$, then the rth terminal of G is a new vertex; Nevertheless, this situation will not occur in our construction). Every terminal of G_c is identified to at most one terminal of G, i.e., each non-zero value in $\{1, \dots, \tau(G_c)\}$ appears at most once in the column c of $m(f)$.

A simple but crucial observation is that the number of possible different matrices, and hence of different composition operations f, is bounded by a function of w. Indeed the size of each matrix is at most $2w$, and each entry of the matrix is an integer between 0 and w.

[1] The definition of [5] considers composition operations of arbitrary arity, i.e., they consider gluing on three or more graphs simultaneously, and they also consider a special gluing, on a single graph which allows to "forget" some terminals. Technically, all these operations can be replaced by operations of arity 2, and we only use arity 2 for the sake of simplifying the presentation.

[2] The figure is borrowed from [33] with the agreement of the authors.

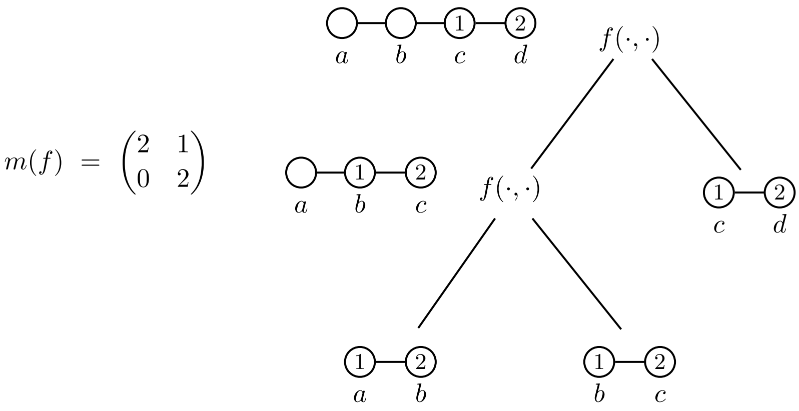


Figure 2 Paths as 2-terminal recursive graphs.

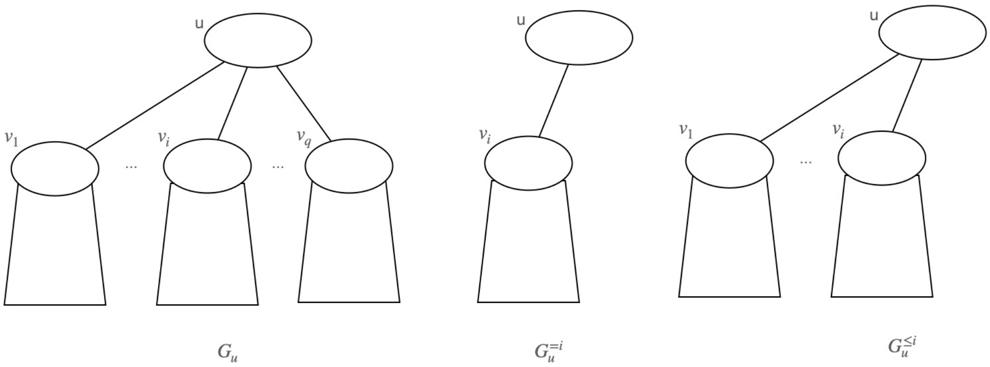


Figure 3 Tree-decompositions: graphs G_u, $G_u^{=i}$ and $G_u^{\leq i}$.

The class of w-terminal recursive graphs is exactly the class of graphs of treewidth at most $w-1$ (see, e.g., Theorem 40 in [4]).

Let us briefly describe how a tree-decomposition of width $w-1$ of a graph G can be transformed into a description of G as a w-terminal recursive graph. This construction will be crucial for efficiently deciding MSO properties of graph G. Let $T=(I,F)$ be a tree-decomposition of $G=(V,E)$ with bags of size at most w. The terminals correspond to the root bag. For every node u of T, we use the following notations, depicted in Figure 3:

- T_u is the subtree of T rooted at u;
- B_u is the bag of node u, and $G_u^{\mathsf{base}} = (G[B_u], B_u)$ is the w-terminal recursive base graph induced by bag B_u;
- V_u is the union of all bags of T_u, and $G_u = (G[V_u], B_u)$ is the w-terminal graph induced by V_u, with B_u as set of terminals.

Let us now show that G_u is indeed a w-terminal recursive graph. This is clear when u is a leaf, since, in this case, $G_u = G_u^{\mathsf{base}}$ is a base graph. Assume that u is not a leaf, and let $v_1, \ldots, v_q$ be the children of node u in T. The ordering of the children is arbitrary, but fixed. Let us introduce two new families of w-terminal recursive graphs as follows. Both are having B_u as set of terminals, and, for every $i \in \{1 \ldots, q\}$:

- $G_u^{=i} = G[B_u \cup V_{v_i}]$, and
- $G_u^{\leq i} = G[B_u \cup V_{v_1} \cup \cdots \cup V_{v_i}]$.

Observe that $G_u^{=i}$ is obtained by gluing G_{v_i} with the base graph G_u^{base}. More precisely,

$$G_u^{=i} = f_{(B_{v_i},B_u)}(G_{v_i}, G_u^{\mathsf{base}}), \tag{1}$$

where the gluing operation $f(B_{v_i}, B_u)$ glues the terminals of $B_{v_i} \cap B_u$ of G_{v_i} to the corresponding terminals of B_u, and the new set of terminals is B_u. Also, for all $i \in \{1, \dots, q-1\}$, $G_u^{\leq i+1}$ is obtained by gluing $G_u^{\leq i}$ and $G_u^{=i+1}$ using the gluing function $f_{(B_u,B_u)}$ (which is the identity function on $|B_u|$ terminals), that is:

$$G_u^{\leq i+1} = f_{(B_u,B_u)}(G_u^{\leq i}, G_u^{=i+1}). \tag{2}$$

By construction, we get $G_u = G_u^{\leq q}$.

4 MSO logic and Courcelle's theorem

Recall that, using monadic second-order (MSO) logic formulas on graphs, we can express graph properties such as "G is not 3-colorable" or "G contains no triangles". In order to solve optimization problems, we also consider MSO formulas with a free variable. That is, we consider formulas of the form $\varphi(S)$ where S is a set of vertices, or a set of edges. The corresponding optimization problem aims at finding a set S with maximum (or minimum, depending on the context) size satisfying $G \models \varphi(S)$. We may even assume that the vertices, or edges of the input graph $G = (V, E)$ have *polynomial* weights, that is the weight assignment $\mathsf{w} : V \cup E \to \mathbb{Z}$ satisfies that, for every $x \in V \cup E$, $w(x)$ can be encoded with $O(\log n)$ bits. The problem $max\varphi$ then consists in computing the set S with maximum weight satisfying $G \models \varphi(S)$. In this framework we can express problems like maximum (weighted) independent set, minimum (weighted) dominating set, or minimum-weight spanning tree (MST).

4.1 Regular Predicates, Homorphism Classes, and Composition

To start, let us first consider closed formulae only, i.e., with no free variable, and formulas with just one free (edge or vertex) set variable. (Further extensions are discussed in Section 6, with applications to labeled graphs and counting.) Using closed formulae, we can refer to graph predicates $\mathcal{P}(G)$, and, using formulas with free variables, we can refer to graph predicates $\mathcal{P}(G, X)$, where X denotes a subset of vertices or a subset of edges of G. For each possible assignment of X with corresponding values, $\mathcal{P}$ is either true or false.

Any composition operation f over two w-terminal recursive graphs $G_1 = (V_1, W_1, E_1)$, and $G_2 = (V_2, W_2, E_2)$ naturally extends to a composition over pairs (G_1, X_1), and (G_2, X_2). If $G = f(G_1, G_2)$, we denote by $\circ_f$ the composition over pairs. More precisely,

$$\circ_f\big((G_1, X_1), (G_2, X_2)\big) = (G, X),$$

the operation being valid only under some specific conditions. Let us consider the case when X_1 and X_2 are vertex-set variables. For each terminal t of G, if terminals from both G_1 and G_2 were mapped to t, say, terminals $t_1 \in W_1$ and $t_2 \in W_2$, then either $t_1 \in X_1$ and $t_2 \in X_2$, or $t_1 \notin X_1$ and $t_2 \notin X_2$. The set X is interpreted as the union of X_1 and X_2, by identifying pairs of terminal vertices $t_1 \in X_1$ and $t_2 \in X_2$ that have been mapped on a same terminal of G. For edge-sets, the set X can also be seen as the union of two sets X_1 and X_2, up to gluing the vertices specified by f. We refer to [5] for a description of the gluing operation, and of the interpretation of the values of the variables.

▶ **Definition 6** (regular predicate). *A graph predicate $\mathcal{P}(G, X)$ is* regular *if, for any value w, we can associate to w*

- *a finite set $\mathcal{C}$ of* homomorphism classes*,*

- *an* homomorphism function h*, assigning to each* w*-terminal recursive graph* G*, and to any subset* X *of vertices or edges of* G*, a* class $h(G, X) \in \mathcal{C}$*, and*
- *an* update function $\odot_f : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ *for each composition operation* f*,*

such that:

1. *If* $h(G_1, X_1) = h(G_2, X_2)$ *then* $\mathcal{P}(G_1, X_1) = \mathcal{P}(G_2, X_2)$*;*
2. *For any two* w*-terminal recursive graphs* G_1 *and* G_2*, and any two sets* X_1 *and* X_2*,*

$$h\Big(\circ_f \left((G_1, X_1), (G_2, X_2)\right)\Big) = \odot_f \big(h(G_1, X_1), h(G_2, X_2)\big).$$

A class $c \in \mathcal{C}$ is said to be an *accepting class* if there exists (G, X) such that $h(G, X) = c$, and $\mathcal{P}(G, X)$ is true. By definition, the predicate $\mathcal{P}$ holds for every (G', X') such that $h(G', X') = c$. A non accepting class c is called a *rejecting class*. The same definitions applies to predicates $\mathcal{P}(G)$, with no free variables.

Remark. Without loss of generality, we may assume that, in Definition 6, the class $c = h(G, X)$ with $G = (V, W, E)$ encodes the intersection of X with $G[W]$. Indeed, since W is of constant size, if X is a vertex-set, then we can add the set of all the ranks of the elements in $X_j \cap W$, with respect to the totally ordered set W, to the class c. And if X_j is an edge-set, then we can store each edge of X_j contained in $G[W]$ as the pair of ranks of its endpoints. In particular, we can assume that we are given a function $\mathsf{Selected}(c, W)$ which, given a class c, and a set of terminals W, returns the unique intersection of X with the vertices, or the edges, of $G[W]$.

▶ **Theorem 7** ([5])**.** *Any predicate* $\mathcal{P}(G, X)$ *expressible by an MSO formula* $\varphi(X)$ *is regular. Moreover, given the formula* $\varphi(X)$ *and a parameter* w*, one can compute the set of classes* $\mathcal{C}$*, the update functions* $\odot_f$ *over all possible composition operations* f*, as well as the homomorphism classes* $h(G, X)$ *for all base graphs* G*, and all possible values of variable* X*. (The same holds for predicates* $\mathcal{P}(G)$ *corresponding to closed formulas* φ*.)*

Let us emphasize that the width parameter w, and the formula φ in Theorem 7 are *constants*. Thereofore, the set of homomorphism classes $\mathcal{C}$ is of constant size, and can be computed, as well as functions $\odot_f$ and homomorphism classes of base graphs, in *constant* time. This constant just depends on w and on φ.

4.2 Sequential Model-Checking and optimization

We have all ingredients to describe the sequential model-checking algorithm on graphs of bounded treewidth, that we will later use for designing our distributed protocol. In a nutshell, the algorithm proceeds by dynamic programming, from the leaves of the decomposition-tree to the root. When considering a node u, the program deals with the graph G_u induced by all bags in the subtree rooted at u, which is viewed as a w-terminal recursive graph with labels B_u. The program computes the homomorphism class of $h(G_u)$ using merely the homomorphism classes of its children, the bags of its children, and the subgraph of G induced by the bag of u.

▶ **Lemma 8** (bottom-up decision)**.** *Let* $\mathcal{P}(G)$ *be a regular predicate on graphs, corresponding to a formula* φ *with no free variables. Let* G *be a graph, and let* $T = (I, F)$ *be a tree-decomposition of* G *with bags* $\{B_u \mid u \in I\}$*. Let* u *be a node of the tree decomposition, with children* $v_1, \dots, v_q$ *for* $q \geq 0$*. The homomorphism class of* $h(G_u)$ *can be computed using only* G_u^{base}*, the values of* B_{v_i} *and* $h(G_{v_i})$ *for all* $i \in \{1, \dots, q\}$*.*

Proof. Observe that $h(G_u^{\mathsf{base}})$ can be computed directly by Theorem 7. In particular this settles the case when u is a leaf. If u is an internal node, then, for each $i = 1, \ldots, q$, we can compute $h(G_u^{=i})$ using $B_{v_i}, B_u, h(G_{v_i})$ and $h(G_u^{\mathsf{base}})$ as follows. By Equation 1, we have $G_u^{=i} = f_{(B_{v_i}, B_u)}(G_{v_i}, G_u^{\mathsf{base}})$. Since B_{v_i} and B_u are known, one can construct the function $f_{(B_{v_i}, B_u)}$, and, thanks to Theorem 7, one can retrieve the function $\odot_{f_{(B_{v_i}, B_u)}}$. By Definition 6, $h(G_u^{=i}) = \odot_{f_{(B_{v_i}, B_u)}}(h(G_{v_i}), h(G_u^{\mathsf{base}}))$. Since the parameters on the right-hand side of the equality are known, one can compute $h(G_u^{=i})$.

Let us now show how to compute the values $h(G_u^{\leq i})$. For $i = 1$, $G_u^{\leq 1} = G_u^{=1}$, and thus $h(G_u^{\leq 1}) = h(G_u^{=1})$. For every $i \geq 2$, one can compute $h(G_u^{\leq i})$ using B_u, $h(G_u^{\leq i-1})$, and $h(G_u^{=i})$. Indeed, by Equation 2, $G_u^{\leq i} = f_{(B_u, B_u)}(G_u^{\leq i-1}, G_u^{=i})$, so again we have $h(G_u^{\leq i}) = \odot_{f_{(B_u, B_u)}}(h(G_u^{\leq i-1}), h(G_u^{=i}))$, and all parameters on the right-hand side have been computed previously.

Eventually, since $G_u = G_u^{\leq q}$, we get $h(G_u) = h(G_u^{\leq q})$. ◀

Algorithm 1 simultaneously presents the model-checking of regular predicates on graphs, and the optimization protocol for predicates on graphs and sets.

For the decision problem, we simply compute bottom-up, for each node u, the class of $h(G_u)$ using Lemma 8. At the root r, the algorithm accepts if $h(G_r)$ is an accepting class.

Algorithm 1 Sequential decision for regular property $\mathcal{P}$ on graph G.

```
Require: tree decomposition $T = (V_E, F_T)$ of width $w - 1$ of $G$ ; formula $\varphi$ and the
    corresponding homomorphism classes $\mathcal{C}$, homomorphism function $h$ on base graphs,
    composition functions $\odot_f$                                    ▷ See Theorem 7

Bottom-up phase on tree $T$, computes $h(G_u)$ for each node $u$ :
for each node $u$ of $T$ from the bottom to the root do
    let $v_1, \ldots, v_q$ be the children of $u$ in $T$     ▷ These nodes have been treated before $u$
    Compute $h(G_u)$ from $G_u^{\mathsf{base}}$, $B_{v_i}$ and $h(G_{v_i})$, $1 \leq i \leq q$ using Lemma 8
end for

Decision at the root $r$ :
Return true if $h(G_r)$ is an accepting class, otherwise return false.
```

A similar result can be obtained for regular predicates $\mathcal{P}(G, X)$, over (polynomially weighted) graphs and vertex sets. In order to compute the maximum or minimum weight set X such that $\mathcal{P}(G, X)$ holds, we can compute by dynamic programming, at each node u of the decomposition tree, a table $\mathsf{OPT}(G_u)$ with $|\mathcal{C}|$ entries, one for each homomorphism class $c \in \mathcal{C}$ of $\mathcal{P}$, where $\mathsf{OPT}(G_u)[c]$ corresponds to the optimum size of a partial solution set X_u such that $\mathcal{P}(G_u, X_u) = c$. This solution X_u is obtained by glueing partial solutions over the children nodes of u, and function $\mathsf{Selected}$ is used to avoid overcounting vertices in the intersection of partial solutions; see the full version of the paper [28] for complete details.

5 Distributed construction of the elimination tree

Our $\mathsf{CONGEST}$ protocol constructing an elimination tree of depth smaller than 2^d for graphs of treedepth at most d is depicted in Algorithm 2. A similar approach was previously used in in [59] for so-called small treedepth decompositions (see also Section 7 of this article).

Algorithm 2 CONGEST algorithm computing an elimination tree of G in $O(2^{2d})$ rounds.

```
Require: Protocol leader($G = (V, E), U$) with a set $U \subseteq V$ of distinguished vertices (i.e.,
      each vertex knows whether it belongs to $U$). After $O(\mathsf{diam}(G))$ CONGEST rounds, each
      vertex $u$ will know leader($u$), the minimum identifier in the component of $G[U]$ containing
      $u$.
 1: Apply leader on all vertices of $G$
 2: Let $r$ be the unique node such that $r =$ leader($r$)            ▷ Unique since $G$ is connected
 3: Set parent($r$) $= r$                                                ▷ $r$ is the root of the tree
 4: Mark vertex $r$                         ▷ Marked vertices are those already placed in the tree
 5: Set depth($r$) $= 1$
 6: for step $i = 2$ to $D = 2^d - 1$ do
 7:     // At step $i$ we identify the nodes of $T$ of depth $i$
 8:     Apply leader on all unmarked vertices of $G$                               ▷ $O(2^d)$ rounds
 9:     Each unmarked vertex $u$ broadcasts (leader($u$), $u$) to its neighbours    ▷ One round
10:     for each marked vertex $v$ of depth $i$ do                            ▷ All in a same round
11:         for each $\ell$ among values leader($u$) received by $v$ do
12:             $v$ picks the corresponding $u(\ell)$ of minimum id ▷ $u(\ell)$ is a new node of depth $i$
13:             $v$ adds $u(\ell)$ to the list of its children
14:             $v$ sends to $u(\ell)$ a message with its id indicating that it becomes its parent.
15:         end for
16:     end for
17:     for each vertex $u$ that receives such a message from some $v$ do ▷ All in a same round
18:         $u$ sets parent($u$) $= v$, depth($u$) $= i$ and marks itself
19:     end for
20: end for
21: if some vertex $u$ is still unmarked then
22:     $u$ rejects because td($G$) $> d$            ▷ $G$ contains a path with more than $D$ vertices
23: end if
```

▶ **Lemma 9.** *Let $G = (V, E)$ be the (connected) input network, and let $d \geq 1$ be an integer. There exists an algorithm performing in $O(2^{2d})$ rounds in* CONGEST *that outputs either an elimination tree $T = (T, F)$ of G with depth at most 2^d, or reports that* $\mathsf{td}(G) > d$. *In the former case, each node $u \in V$ knows its parent and its children in the tree T at the end of the algorithm, as well as the depth of T.*

Proof. Algorithm 2 constructs an elimination tree following the same approach as Lemma 2, in a greedy manner: in a nutshell, it computes a depth-first search tree of G, heavily relying on the fact that, if G is of treedepth at most d, then all its paths are of length at most 2^d. Since G is connected, it starts with a root vertex $v = r$ (chosen arbitrarily), and then constructs elimination trees of $G \setminus v$, by treating each component separately. The components of $G \setminus v$ are identified by their leader with the smallest node's identifier of the component. Each unmarked node eventually knows its leader (Instruction 8). For a component with leader ℓ, we choose as root of the component a vertex that is adjacent to v (Instruction 12). In particular, every edge of the tree is also an edge of G (see Instruction 14).

The construction preserves the following invariant. The tree constructed after step i is an elimination tree of the subgraph induced by the marked vertices. Moreover, for each connected component of unmarked vertices, its outgoing edges are solely incident to a path from the root and a vertex v of depth i. In particular, at the end, T is an elimination tree of G. Furthermore, T is a subtree of G. Therefore, by Lemma 5, if $\mathsf{td}(G) \leq d$ then the depths of T is smaller than 2^d as requested, and the algorithm marks all vertices in less than 2^d phases. Consequently, if some vertices remain unmarked after this many rounds (Instruction 21), we correctly assert that $\mathsf{td}(G) > d$.

Regarding the round-complexity, observe that, at each step, there is a call to Algorithm leader on the set of unmarked nodes (see, e.g., [48] for a detailed description of a leader-election algorithm). Its round complexity is $O(\mathsf{diam}(G))$. The diameter of G is $O(2^{\mathsf{td}(G)})$, and thus is it at most $O(2^d)$ (we can adapt algorithm leader such that, if it does not succeed in $O(2^d)$ rounds, then it rejects, which is correct as, in this case, $\mathsf{td}(G) > d$). ◀

▶ **Lemma 10.** *Let $G = (V, E)$ be the input network, and let us assume that an elimination tree $T = (F, V)$ of depth smaller than 2^d has been constructed as in Lemma 9. There is a CONGEST algorithm constructing the canonical tree decomposition $(T, (B_u)_{u\in V})$ in $O(2^d)$ rounds. At the end of the algorithm, each node u knows its bag B_u as well as the graph $G[B_u]$ induced by the bag.*

Proof. The algorithm proceeds top-down. For each round $i = 1, \dots, D = 2^d - 1$, every node u at depth i computes B_u and $G[B_u]$. Observe that when u is the root, B_u is a singleton so $G[B_u]$ is trivial. If u is not the root, then u has received B_v and $G[B_v]$ from its parent v. Observe that $B_u = B_v \cup \{u\}$ and the edges of $G[B_u]$ are the edges of $G[B_v]$, plus the edges incident to u. Therefore, node u is able to compute the information from its parent, and to transmit it to its children. ◀

6 Distributed model checking and optimization

We have now all ingredients to prove our main result.

▶ **Theorem 11** (Distributed decision and optimization).

- *For any closed MSO formula φ, there exists an algorithm which, for any n-node graph G, and any $d \geq 0$, decides whether $G \models \varphi$, or reports "large treedepth" if $\mathsf{td}(G) > d$, running in $O(2^{2d})$ rounds in the CONGEST model.*
- *For any MSO formula $\varphi(S)$ with a free variable S representing a vertex-set, or an edge-set, there exists an algorithm which, for any n-node graph G, and any $d \geq 0$, selects a set S of maximum weight satisfying $G \models \varphi(S)$, or reports "large treedepth" if $\mathsf{td}(G) > d$, running in $g(d, \varphi)$ rounds in the CONGEST model for some function g.*

Proof. By Lemmas 9 and 10, one can construct a canonical tree decomposition $T = (V, F)$ of $G = (V, E)$, with bags $\{B_u \mid u \in V\}$ of width at most 2^d (or correctly reject because $\mathsf{td}(G) > d$), in $O(2^{2d})$ rounds. Moreover each node u knows its parent $\mathsf{parent}(u)$, its bag B_u, the graph $G[B_u]$, and its depth in T. By construction, the tree T is a subgraph of G. It remains to show that, based on these elements, one can implement the sequential algorithm (cf. Algorithm 1) in CONGEST.

Let us first consider model-checking of a closed formula φ. We describe how the bottom-up phase, and the decision at the root in Algorithm 1 can be implemented in $\mathsf{depth}(T)$ rounds. Let us consider all steps $j \in \{1, \dots, \mathsf{depth}(T)\}$, where each step consists of a single round. At step j, all nodes u of depth $k = \mathsf{depth}(T) - j + 1$ can compute the homomorphism classes $h(G_u)$ in parallel (k decreases from $\mathsf{depth}(T)$ to 1), and can send the results of this computation to their parents. Indeed, if u of depth k is a leaf, then it has all information needed to compute $h(G_u)$ already, because it only needs to know graph $G_u^{\mathsf{base}} = G[B_u]$ (see Instruction 4 of Algorithm 1, and Lemma 8). If u it is not a leaf, then it also needs the bags

B_{v_i}, and the homomorphism classes $h(G_{v_i})$ from all its children v_i, $\{1,\dots,q\}$. But, at step j, node u has precisely already received these information from its children, who have sent them at the previous step $j-1$. The decision at the root can be performed at round 1. The root accepts or rejects depending on its homomorphism class, as in Algorithm 1, and all other nodes accept. Therefore, if $G \models S$, then all nodes accept, otherwise the root rejects. Note that each message consists of a homomorphism class, thus the size of the messages is a constant. More precisely, messages are of size $\log|\mathcal{C}|$ bits, where $\mathcal{C}$ denotes the set of homomorphism classes for property φ and treedepth at most 2^d.

For the optimization version, due to space restriction the details are given in the full version [28]. In a nutshell, the bottom up phase computes at each node u a table $\mathsf{OPT}(G_u)$ of size $\mathcal{C}$ which, for each homomorphism class $c \in \mathcal{C}$, stores the size of the optimal partial solution S_u corresponding to that class. Therefore the round complexity depends on the size of this table OPT, which is upper bounded by some function g depending on d and formula φ. A top-down phase is needed in order to mark the vertices of the global optimal solution. ◀

7 Applications to H-freeness for graphs of bounded expansion

For the many alternative definitions of graphs of bounded expansion, we refer to the book of Nešetřil and Ossona de Mendez in [58]. In terms of applications, we simply recall that the class of planar graphs, and, more generally, every class of graphs excluding a fixed minor, are classes of graphs of bounded expansion. It is known that graphs of bounded expansion admit so-called *low treedepth decompositions.*

▶ **Theorem 12** ([58]). *Let $\mathcal{G}$ be a class of graphs of bounded expansion. There is a function $f : \mathbb{N} \to \mathbb{N}$ such that, for every integer $p > 0$, and every graph $G = (V, E) \in \mathcal{G}$, the vertex set of G can be partitioned into at most $f(p)$ parts $V_1, \dots, V_{f(p)}$ such that the union of any q parts, $1 \le q \le p$, induces a subgraph of G with treedepth at most q.*

A partition satisfying the property of Theorem 12 is called a *low treedepth decomposition* of G for parameter p. Interestingly, low treedepth decompositions can be efficiently computed in CONGEST, i.e., each vertex can compute the index $i \in \{1,\dots,f(p)\}$ of the part to which it belongs.

▶ **Theorem 13** ([59]). *For every graph class $\mathcal{G}$ with bounded expansion, and every positive integer p, a low treedepth decomposition of G for parameter p can be computed in $O(\log n)$ rounds in* CONGEST.

The constant hidden in the big-O notation in the statement of Theorem 13 depends on the class $\mathcal{G}$ and on the parameter p, and it is quite huge. The proof of Theorem 13 is sophisticated, but the algorithm is actually quite simple. It is merely based on the fact that graphs with bounded expansion have bounded degeneracy, and on the use of standard distributed tools for approximating the degeneracy of a graph in CONGEST. Combining Theorem 11 with Theorem 13, we can establish the following.

▶ **Corollary 14.** *Let $\mathcal{G}$ be a class of graphs with bounded expansion, and let H be a connected graph. Deciding H-freeness for graphs in $\mathcal{G}$ can be achieved $O(\log n)$ rounds under the* CONGEST *model.*

Proof. The algorithm works as follows. Let p be the number of vertices of H. First, compute a low treedepth decomposition $V_1, \dots, V_{f(p)}$ of the input graph $G = (V, E)$ into $f(p)$ parts for parameter p using Theorem 13. Then, for every non-empty set $I \subseteq [f(p)]$ with $|I| \le p$, let $G_I = G[\cup_{i\in I} V_i]$ be the graph induced by the parts V_i, $i \in I$. Note that there are at most $\binom{f(p)}{p}$ such subsets I, that is, a constant number of choices for I. Also observe that if a copy of H exists in graph G, then this copy of H belongs to at least one of the graphs G_I. Indeed, this holds for every set I of at most p parts V_i such that each of the p vertices of the copy of H is contained in some part V_i. Informally, we "guess" the colors of the copies of H, and I must contain these at most p colors. It is therefore sufficient to run the algorithm in Theorem 11 on each graph G_I in parallel, and to reject if one of the parallel executions finds a copy of H. This is doable because (1) G_I is of treedepth at most p, (2) if a copy of H exists, then it will be found in a connected component of G_I, thanks to the fact that H is connected, and (3) the property "G_I is H-free" can be expressed as an MSO formula (actually, even as an FO formula), with p variables, one for each vertex of H. For instance, for a graph $H = (V_H, E_H)$ with $V_H = \{1, 2, \dots, p\}$, we can use the formula

$$\varphi_H = \neg\exists x_1, x_2, \dots, x_p \left(\bigwedge_{\{i,j\}\in E_H} \mathsf{adj}(x_i, x_j) \right) \wedge \left(\bigwedge_{\{i,j\}\notin E_H} \neg\mathsf{adj}(x_i, x_j) \right).$$

Thus the algorithm rejects if and only if the input graph contains a copy of H, as desired. ◀

In particular, Corollary 14 proves that H-freeness can be solved in $O(\log n)$ rounds in planar graphs under CONGEST. In contrast, for arbitrary graphs, even C_4-freeness requires $\Omega(\sqrt{n})$ rounds, and, for every $p \geq 2$, there are $O(p)$-vertex graphs H for which H-freeness requires $\Omega(n^{2-1/p})$ rounds [27]. Note that H-freeness can be considered in the usual sense (i.e., the input graph does not contain any copy of H as an induced subgraph), but also in the mere sense that there are no copies of H as a (non necessarily induced) subgraphs, by a straightforward adaptation of the MSO formula describing the problem.

8 Conclusion

In this paper, we established a meta-theorem about MSO formulas on graphs with bounded treedepth within the CONGEST model. Treedepth plays a fundamental role in the theory of sparse graphs of Nešetřil and Ossona de Mendez [58]. In particular, decomposing a graph in graphs of bounded treedepth is the crucial step in deriving a linear-time model-checking algorithm for FO on graphs of bounded *expansion* in the sequential computational model. Graphs of bounded expansion contain bounded-degree graphs, planar graphs, graphs of bounded genus, graphs of bounded treewidth, graphs that exclude a fixed minor, etc. Model-checking for FO on graphs of bounded expansion cannot be achieved in the CONGEST model since, as we already mention, checking an FO predicate as simple as "there is at least one vertex of degree > 2" requires $\Omega(n)$ rounds in n-node trees. Nevertheless, there might exist some fragments of FO that could be tractable on graphs of bounded expansion in the distributed setting. It would be interesting to identify the exact boundaries of intractability in this context, regarding both distributed decision, and distributed certification. An initial step in this direction was taken by Nešetřil and Ossona de Mendez in [59], resulting in a distributed algorithm for computing a low treedepth decomposition of graphs of bounded expansion, running in $O(\log n)$ rounds under CONGEST. As we illustrated, this results allows to efficiently decide FO-expressible decision problems (such as H-freeness, for H connected) for classes of graphs with bounded expansion, in $O(\log n)$ rounds. We restate the open question of [59]: Given a *local* FO formula $\varphi(x)$, i.e., a formula where $\varphi(x)$ depends on a fixed-radius neighborhood of vertex x only, can we mark all vertices satisfying φ in $O(\log n)$ rounds?

References

1 Alkida Balliu, Gianlorenzo D'Angelo, Pierre Fraigniaud, and Dennis Olivetti. What can be verified locally? *J. Comput. Syst. Sci.*, 97:106–120, 2018. doi:10.1016/J.JCSS.2018.05.004.

2 Aviv Bick, Gillat Kol, and Rotem Oshman. Distributed zero-knowledge proofs over networks. In *33rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2426–2458, 2022. doi:10.1137/1.9781611977073.97.

3 Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder. Optimal space lower bound for deterministic self-stabilizing leader election algorithms. *Discret. Math. Theor. Comput. Sci.*, 25:1–17, 2023. doi:10.46298/DMTCS.9335.

4 Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1-2):1–45, 1998. doi:10.1016/S0304-3975(97)00228-4.

5 Richard B. Borie, R. Gary Parker, and Craig A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7(5&6):555–581, 1992. doi:10.1007/BF01758777.

6 Nicolas Bousquet, Laurent Feuilloley, and Théo Pierron. Local certification of graph decompositions and applications to minor-free classes. In *25th International Conference on Principles of Distributed Systems (OPODIS)*, volume 217 of *LIPIcs*, pages 22:1–22:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.OPODIS.2021.22.

7 Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. Distributed connectivity decomposition. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 156–165, 2014. doi:10.1145/2611462.2611491.

8 Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof-labeling schemes. *Theor. Comput. Sci.*, 811:112–124, 2020. doi:10.1016/J.TCS.2018.08.020.

9 Yi-Jun Chang. Efficient distributed decomposition and routing algorithms in minor-free networks and their applications. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, pages 55–66, 2023. doi:10.1145/3583668.3594604.

10 Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990. doi:10.1016/0890-5401(90)90043-H.

11 Pierluigi Crescenzi, Pierre Fraigniaud, and Ami Paz. Trade-offs in distributed interactive proofs. In *33rd International Symposium on Distributed Computing (DISC)*, volume 146 of *LIPIcs*, pages 13:1–13:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.DISC.2019.13.

12 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012. doi:10.1137/11085178X.

13 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *33rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 367–376, 2014. doi:10.1145/2611462.2611493.

14 Michael Elberfeld, Martin Grohe, and Till Tantau. Where first-order and monadic second-order logic coincide. *ACM Trans. Comput. Log.*, 17(4):25, 2016. doi:10.1145/2946799.

15 Gábor Elek. Planarity can be verified by an approximate proof labeling scheme in constant-time. *J. Comb. Theory, Ser. A*, 191:105643, 2022. doi:10.1016/J.JCTA.2022.105643.

16 Yuval Emek, Yuval Gil, and Shay Kutten. Locally restricted proof labeling schemes. In *36th International Symposium on Distributed Computing (DISC)*, volume 246 of *LIPIcs*, pages 20:1–20:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.DISC.2022.20.

17 Louis Esperet and Benjamin Lévêque. Local certification of graphs on surfaces. *Theor. Comput. Sci.*, 909:68–75, 2022. doi:10.1016/J.TCS.2022.01.023.

18 Louis Esperet and Sergey Norin. Testability and local certification of monotone properties in minor-closed classes. In *49th International Colloquium on Automata, Languages, and

Programming (ICALP), volume 229 of *LIPIcs*, pages 58:1–58:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ICALP.2022.58.

19 Laurent Feuilloley. Introduction to local certification. *Discret. Math. Theor. Comput. Sci.*, 23(3):1–23, 2021. doi:10.46298/DMTCS.6280.

20 Laurent Feuilloley, Nicolas Bousquet, and Théo Pierron. What can be certified compactly? compact local certification of MSO properties in tree-like graphs. In *41st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, 2022. doi:10.1145/3519270.3538416.

21 Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. *Bull. EATCS*, 119, 2016. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/411.

22 Laurent Feuilloley and Pierre Fraigniaud. Error-sensitive proof-labeling schemes. *J. Parallel Distributed Comput.*, 166:149–165, 2022. doi:10.1016/J.JPDC.2022.04.015.

23 Laurent Feuilloley, Pierre Fraigniaud, and Juho Hirvonen. A hierarchy of local decision. *Theor. Comput. Sci.*, 856:51–67, 2021. doi:10.1016/J.TCS.2020.12.017.

24 Laurent Feuilloley, Pierre Fraigniaud, Juho Hirvonen, Ami Paz, and Mor Perry. Redundancy in distributed proofs. *Distributed Comput.*, 34(2):113–132, 2021. doi:10.1007/S00446-020-00386-Z.

25 Laurent Feuilloley, Pierre Fraigniaud, Pedro Montealegre, Ivan Rapaport, Éric Rémila, and Ioan Todinca. Compact distributed certification of planar graphs. *Algorithmica*, 83(7):2215–2244, 2021. doi:10.1007/S00453-021-00823-W.

26 Laurent Feuilloley, Pierre Fraigniaud, Pedro Montealegre, Ivan Rapaport, Éric Rémila, and Ioan Todinca. Local certification of graphs with bounded genus. *Discret. Appl. Math.*, 325:9–36, 2023. doi:10.1016/J.DAM.2022.10.004.

27 Orr Fischer, Tzlil Gonen, Fabian Kuhn, and Rotem Oshman. Possibilities and impossibilities for distributed subgraph detection. In Christian Scheideler and Jeremy T. Fineman, editors, *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 153–162. ACM, 2018. doi:10.1145/3210377.3210401.

28 Fedor V. Fomin, Pierre Fraigniaud, Pedro Montealegre, Ivan Rapaport, and Ioan Todinca. Distributed model checking on graphs of bounded treedepth. *CoRR*, abs/2405.03321, 2024. doi:10.48550/arXiv.2405.03321.

29 Pierre Fraigniaud, François Le Gall, Harumichi Nishimura, and Ami Paz. Distributed quantum proofs for replicated data. In *12th Innovations in Theoretical Computer Science Conference (ITCS)*, volume 185 of *LIPIcs*, pages 28:1–28:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ITCS.2021.28.

30 Pierre Fraigniaud, Mika Göös, Amos Korman, Merav Parter, and David Peleg. Randomized distributed decision. *Distributed Comput.*, 27(6):419–434, 2014. doi:10.1007/S00446-014-0211-X.

31 Pierre Fraigniaud, Mika Göös, Amos Korman, and Jukka Suomela. What can be decided locally without identifiers? In *32nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 157–165, 2013. doi:10.1145/2484239.2484264.

32 Pierre Fraigniaud, Amos Korman, and David Peleg. Towards a complexity theory for local distributed computing. *J. ACM*, 60(5):35:1–35:26, 2013. doi:10.1145/2499228.

33 Pierre Fraigniaud, Frédéric Mazoit, Pedro Montealegre, Ivan Rapaport, and Ioan Todinca. Distributed certification for classes of dense graphs. In *37th International Symposium on Distributed Computing (DISC)*, volume 281 of *LIPIcs*, pages 20:1–20:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.DISC.2023.20.

34 Pierre Fraigniaud, Pedro Montealegre, Ivan Rapaport, and Ioan Todinca. A meta-theorem for distributed certification. *Algorithmica*, 86(2):585–612, 2024. doi:10.1007/S00453-023-01185-1.

35 Pierre Fraigniaud, Boaz Patt-Shamir, and Mor Perry. Randomized proof-labeling schemes. *Distributed Comput.*, 32(3):217–234, 2019. doi:10.1007/S00446-018-0340-8.

36 Jakub Gajarský and Petr Hlinený. Kernelizing MSO properties of trees of fixed height, and some consequences. *Log. Methods Comput. Sci.*, 11(1):1–26, 2015. doi:10.2168/LMCS-11(1:19)2015.

37 François Le Gall, Masayuki Miyamoto, and Harumichi Nishimura. Distributed quantum interactive proofs. In *40th International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 254 of *LIPIcs*, pages 42:1–42:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.STACS.2023.42.

38 JA Garay, S Kutten, and D Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. In *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, pages 659–668. IEEE, 1993.

39 Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. Near-optimal distributed maximum flow. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 81–90, 2015.

40 Mohsen Ghaffari and Fabian Kuhn. Distributed minimum cut approximation. In *International Symposium on Distributed Computing*, pages 1–15. Springer, 2013. doi:10.1007/978-3-642-41527-2_1.

41 Mohsen Ghaffari and Christoph Lenzen. Near-optimal distributed tree embedding. In *International Symposium on Distributed Computing*, pages 197–211. Springer, 2014. doi:10.1007/978-3-662-45174-8_14.

42 Mohsen Ghaffari and Merav Parter. Near-optimal distributed dfs in planar graphs. In *31st International Symposium on Distributed Computing (DISC 2017)*, pages 21:1–21:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.DISC.2017.21.

43 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory Comput.*, 12(1):1–33, 2016. doi:10.4086/TOC.2016.V012A019.

44 Martin Grohe. Logic, graphs, and algorithms. In *Logic and Automata: History and Perspectives, in Honor of Wolfgang Thomas*, volume 2 of *Texts in Logic and Games*, pages 357–422. Amsterdam University Press, 2008. URL: https://eccc.weizmann.ac.il/report/2007/091/.

45 Martin Grohe and Stephan Kreutzer. Methods for algorithmic meta theorems. In *Model Theoretic Methods in Finite Combinatorics - AMS-ASL Joint Special Session*, volume 558, pages 181–206. AMS, 2009. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.395.8282&rep=rep1&type=pdf.

46 Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Low-congestion shortcuts without embedding. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 451–460, 2016. doi:10.1145/2933057.2933112.

47 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 489–498, 2016. doi:10.1145/2897518.2897638.

48 Juho Hirvonen and Jukka Suomela. *Distributed Algorithms 2020*. Aalto University, 2020. URL: https://jukkasuomela.fi/da2020/.

49 Gillat Kol, Rotem Oshman, and Raghuvansh R. Saxena. Interactive distributed proofs. In *37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 255–264. ACM, 2018. URL: https://dl.acm.org/citation.cfm?id=3212771.

50 Amos Korman, Shay Kutten, and Toshimitsu Masuzawa. Fast and compact self-stabilizing verification, computation, and fault detection of an MST. *Distributed Comput.*, 28(4):253–295, 2015. doi:10.1007/S00446-015-0242-Y.

51 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Comput.*, 22(4):215–233, 2010. doi:10.1007/S00446-010-0095-3.

52 Stephan Kreutzer. Algorithmic meta-theorems. In *Finite and Algorithmic Model Theory*, volume 379 of *London Mathematical Society Lecture Note Series*, pages 177–270. Cambridge University Press, 2011. URL: http://www.cs.ox.ac.uk/people/stephan.kreutzer/Publications/amt-survey.pdf.

53 Shay Kutten and David Peleg. Fast distributed construction of k-dominating sets and applications. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 238–251, 1995.

54 Christoph Lenzen and Boaz Patt-Shamir. Improved distributed steiner forest construction. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 262–271, 2014. doi:10.1145/2611462.2611464.

55 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 565–573, 2014. doi:10.1145/2591796.2591850.

56 Danupon Nanongkai and Hsin-Hao Su. Almost-tight distributed minimum cut algorithms. In *International Symposium on Distributed Computing*, pages 439–453. Springer, 2014. doi:10.1007/978-3-662-45174-8_30.

57 Moni Naor, Merav Parter, and Eylon Yogev. The power of distributed verifiers in interactive proofs. In *31st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1096–115. SIAM, 2020. doi:10.1137/1.9781611975994.67.

58 Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity - Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics*. Springer, 2012. doi:10.1007/978-3-642-27875-4.

59 Jaroslav Nešetřil and Patrice Ossona de Mendez. A distributed low tree-depth decomposition algorithm for bounded expansion classes. *Distributed Comput.*, 29(1):39–49, 2016. doi:10.1007/S00446-015-0251-X.

60 David Peleg. *Distributed computing: a locality-sensitive approach.* SIAM, 2000.

61 Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984. doi:10.1016/0095-8956(84)90013-3.

Content-Oblivious Leader Election on Rings

Fabian Frei
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Ran Gelles
Bar-Ilan University, Ramat Gan, Israel

Ahmed Ghazy
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
Saarland University, Saarbrücken, Germany

Alexandre Nolin
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Abstract

In *content-oblivious computation*, n nodes wish to compute a given task over an asynchronous network that suffers from an extremely harsh type of noise, which corrupts the content of all messages across all channels. In a recent work, Censor-Hillel, Cohen, Gelles, and Sela (Distributed Computing, 2023) showed how to perform arbitrary computations in a content-oblivious way in 2-edge connected networks but only if the network has a distinguished node (called *root*) to initiate the computation.

Our goal is to remove this assumption, which was conjectured to be necessary. Achieving this goal essentially reduces to performing a content-oblivious leader election since an elected leader can then serve as the root required to perform arbitrary content-oblivious computations. We focus on ring networks, which are the simplest 2-edge connected graphs. On *oriented* rings, we obtain a leader election algorithm with message complexity $O(n \cdot \mathsf{ID}_{\max})$, where $\mathsf{ID}_{\max}$ is the maximal assigned ID. As it turns out, this dependency on $\mathsf{ID}_{\max}$ is inherent: we show a lower bound of $\Omega(n \log(\mathsf{ID}_{\max}/n))$ messages for content-oblivious leader election algorithms. We also extend our results to *non-oriented* rings, where nodes cannot tell which channel leads to which neighbor. In this case, however, the algorithm does not terminate but only reaches quiescence.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Content-Oblivious Computation, Faulty Communication, Leader Election, Ring Networks, Ring Orientation

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.26

Related Version *Full Version*: `https://arxiv.org/abs/2405.03646` [22]

Funding *Ran Gelles*: Partially supported by a grant from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel, Grant No. 2020277.

Acknowledgements R. Gelles would like to thank CISPA – Helmholtz Center for Information Security and MPI – Max Planck Institute for Informatics for hosting him while part of this work was done.

1 Introduction

The field of distributed computing is rich with models helping us understand different types of architectures and computational hardness by making different kinds of assumptions about how computations are carried out. A recent work by Censor-Hillel, Cohen, Gelles, and Sela [8] introduced a particularly weak computational model coined *fully defective networks*. This model considers an asynchronous network in which messages may be fully corrupted and thus not carry any information beyond their sheer existence. Algorithms designed for this model cannot rely in any way on possible contents of messages, and thus are named *content-oblivious*. Instead of relying on the content of messages or on their time of arrival (since arbitrary delays may occur in asynchronous networks), such algorithms depend solely on the *order* in which messages arrive from different neighbors.

The aforementioned work by Censor-Hillel et al. [8] showed that any computation possible in the asynchronous setting with reliable content-carrying messages can be simulated in the fully defective setting under two assumptions: that the network is 2-edge connected and that there is a distinguished leader (which they call the *root node*) in the network. While the same paper showed 2-edge connectivity to be essential for any kind of nontrivial computation in fully defective networks, the question of the necessity of a pre-existing leader was not settled. Censor-Hillel et al. conjectured that general computations in fully defective networks do indeed require the existence of such a pre-elected leader.

In this paper, we *disprove* this conjecture, at least for the most fundamental type of 2-edge connected topologies, namely, rings. We design a content-oblivious algorithm that successfully performs a leader election in oriented rings.

▶ **Theorem 1.** *There is a* quiescently terminating *content-oblivious algorithm of message complexity* $n(2 \cdot \mathsf{ID}_{\max} + 1)$ *that elects a leader in oriented rings of* n *nodes with unique IDs.*

Here, $\mathsf{ID}_{\max}$ denotes the maximal ID assigned to a node in the network, and *quiescent termination* refers to the valuable property of our algorithm that nodes do not receive messages after termination. We discuss quiescent computations and the issue of composability in more detail in Section 1.1.

We further extend our result to the case where the ring is non-oriented, albeit under a weaker definition of computation. Instead of termination, we only require *stabilization*, which means that every node eventually settles on a decision (to be or not to be a leader, in our case) that is never revised again. However, the nodes might not know whether they have achieved their stable output already and remain ready to receive and potentially send messages forever. This potential is not actualized for a *quiescently* stabilizing algorithm, where all messaging activity ceases after finite time. Our algorithm not only elects a leader in such a manner but also orients the ring.

▶ **Theorem 2.** *There is a quiescently stabilizing content-oblivious algorithm of message complexity* $n(2 \cdot \mathsf{ID}_{\max} + 1)$ *that elects a leader and orients a non-oriented ring of* n *nodes with unique IDs.*

Considering this weaker definition of computation also allows us to perform the same tasks on the *anonymous ring.* In this setting, nodes are not given identifiers and they all have the same initial state, but each of them has access to an independent source of randomness. Furthermore, the algorithm is allowed to reach an incorrect result as long as it does so with sufficiently small probability (below an arbitrary constant negative power of n). It follows from prior work that there is no terminating algorithm for electing a leader in this setting. However, if we require only quiescent stabilization instead of termination, both electing a leader and orienting a ring are possible.

▶ **Theorem 3.** *There is a content-oblivious algorithm of complexity* $n^{O(1)}$ *that elects a leader and orients an anonymous ring of* n *nodes, each with access to its own source of randomness, with high probability. The algorithm reaches quiescence but does not terminate.*

Both of our deterministic leader election algorithms, for oriented and non-oriented rings, have message complexity $O(n \cdot \mathsf{ID}_{\max})$. Since $\mathsf{ID}_{\max} \geq n$, this implies at least cn^2 pulses, for some constant c. The term $\mathsf{ID}_{\max}$ is not very common in message complexities of algorithms, despite a few exceptions [21, 27].[1] Somewhat surprisingly, our analysis shows that this term is inherent to content-oblivious algorithms. We prove the following lower bound.

[1] Some algorithms depend on the IDs assigned to nodes. However, it is common for the complexity analysis to assume that all IDs are of length $O(\log n)$ bits, making the dependence on the IDs implicit.

▶ **Theorem 4.** *Any deterministic terminating content-oblivious algorithm for leader election in rings with unique IDs sends at least* $n\lfloor\log(\mathsf{ID}_{\max}/n)\rfloor$ *messages.*

Note that this $\Omega(n\log(\mathsf{ID}_{\max}/n))$ lower bound implies that the number of messages in a ring of size n is unbounded – we can always increase the message complexity by assigning larger IDs, even when n is a small constant.

1.1 Quiescence and Composability

As mentioned above, we concatenate our content-oblivious leader election algorithm (Theorem 1) with the root-dependent universal content-oblivious algorithm [8, Thm. 1] to obtain the following powerful corollary.

▶ **Corollary 5.** *Assuming unique IDs, any asynchronous algorithm on rings can be simulated in a fully defective oriented ring.*

However, some subtleties arise when concatenating algorithms in the content-oblivious setting. With reliable message content, each message could be tagged, indicating the algorithm it belongs to. However, when concatenating content-oblivious algorithms, messages sent by the first algorithm may be mistaken for ones sent by the second algorithm, and vice versa.

To make this composition work, we require two properties: (1) *termination*, i.e., nodes should have a distinct point in time where they end the first algorithm and switch to the second one, and (2) *message-algorithm attribution*, i.e., while a node executes an algorithm, it only ever receives messages generated by nodes while they were executing the same algorithm.

Our algorithm for oriented rings (Theorem 1) achieves correct message-algorithm attribution (when composed with the scheme of [8]) by combining two mechanisms. First, it features a *quiescent* termination. In particular, any node terminates eventually, and at that time no messages are in delivery towards that node anymore, nor will any message be sent to that node after its termination. Secondly, the nodes terminate in order, so that the leader is the last to terminate. This makes our algorithm easy to compose with the algorithm of [8], by replacing the act of termination with the act of switching to the second algorithm. The leader, which is the last to terminate the first algorithm, is the node that initiates the computation of the scheme in [8] (i.e., it acts as the root), and at the time it sends its first message, we are guaranteed that all other nodes have already switched to that algorithm.

As a matter of fact, quiescent termination could be relaxed when composing general algorithms: If we have a bound r on the number of messages of the first algorithm that might reach a node after it transitions to the second algorithm, we could still concatenate any algorithm in an *altered form* where nodes send $r+1$ copies of each message, and process arriving messages in groups of $r+1$ messages as well. However, this clearly leads to an undesired r-fold increase in the message complexity of the composed algorithm.

On the other hand, our algorithm for non-oriented rings (Theorem 2) does not terminate and cannot be composed with other algorithms. As mentioned above, we only require the algorithm to reach quiescence in this case.

1.2 Related Work

Leader election is a fundamental task that has been studied by the distributed computing community since the 1970s. Simple leader election algorithms for asynchronous rings were proposed by Le Lann [28] and by Chang and Roberts [10]. These algorithms employ $O(n^2)$ messages to ensure that all nodes yield to the node with the maximal ID, which becomes the leader. Later work [25, 14, 29] improved this down to $O(n \log n)$ messages. This complexity is tight in asynchronous rings [7, 21]. In *synchronous* rings, leader election can be performed by communicating only $O(n)$ messages [21, 17].

For the case of anonymous rings, i.e., with identical nodes without IDs, Angluin [2] proved that symmetry cannot be broken, and thus no leader election algorithm exists. Attiya, Snir, and Warmuth [5] examined anonymous asynchronous rings further and characterized computable tasks. In particular, they showed that many tasks, including ring orientation, require communicating $\Omega(n^2)$ messages. Attiya and Snir [4] gave tight bounds on the complexities of randomized algorithms. Relaxed notions of ring orientation were given by Attiya, Snir, and Warmuth [5], where the orientation is either consistent with all nodes or alternating between any two neighbors, and by Syrotiuk and Pachel [31], where nodes determine whether a majority agrees on the same orientation or not. Orienting a ring and leader election when n is known to the nodes is presented by Flocchini et al. [19].

The question of termination in anonymous rings was explored by Itai and Rodeh [26], who discovered that a network cannot compute its size n by a terminating algorithm. As a consequence, a leader election algorithm that terminates is impossible, too, since it is trivial to learn n once a leader is given. On the other hand, if the nodes know n or even an upper bound on n, a randomized leader election algorithm that terminates exists [26]. Afek and Matias [1] give various leader election algorithms that trade off knowledge, termination, and number of sent messages.

Censor-Hillel, Gelles, and Haeupler [9] designed a content-oblivious BFS algorithm as a pre-processing step for their distributed interactive coding scheme [23]. Building on that idea, Censor-Hillel et al. [8] introduced the concept of content-oblivious computation and proved that general computations are only possible over 2-edge connected networks. They designed a compiler that converts any asynchronous algorithm into a content-oblivious one, assuming a pre-existing leader. They also gave explicit algorithms for content-oblivious DFS, ear decomposition, and (generalized) Hamiltonian cycle construction.

Computation with fully corrupted messages has been comparatively more studied in the synchronous setting, where the presence or absence of a message in a given round can still be used to send one bit of information [30]. A notable example is the *Beeping* model, introduced by Cornejo and Kuhn [11], in which each message sent by a node is received by all its neighbors, and nodes can only distinguish between receiving no message and receiving at least one message. Among other problems, leader election has been studied in this setting in a sequence of works [24, 20, 15, 12]. Similar to the fully defective setting, compilers were devised for transforming algorithms with stronger communication primitives into algorithms for the Beeping model [16, 3, 13].

1.3 Organization

Section 2 formally introduces the content-oblivious setting and some other notions and notations. In Section 3 we give our leader election algorithm for oriented rings, deferring some proofs to the full version of our paper [22]. Section 4 discusses non-oriented rings. In Section 5, we discuss anonymous rings and design a randomized quiescently stabilizing algorithm with a high probability of success for both electing a leader and orienting a non-oriented ring, with some proofs deferred to the full version of our paper [22]. Our lower bound on the number of messages sent by a content-oblivious leader election algorithm is presented in Section 6. Section 7 concludes our work with a brief summary and suggests a few follow-up questions.

2 Preliminaries

Notations. For an integer n, we denote the set $\{1, 2, \dots, n\}$ by $[n]$. All logarithms are binary. For a variable *var* and a node v, we let $var[v]$ denote the value the variable *var* holds at the node v. By the term *with high probability* we mean a probability of at least $1 - n^{-O(1)}$.

The content-oblivious computation model. Consider a distributed network $G = (V, E)$ with $n = |V|$ nodes. We usually assume that each node $v \in V$ is assigned a positive integer as its (unless otherwise stated) unique ID, usually denoted by $\mathsf{ID}_v \in \mathbb{N}$. We denote the largest ID assigned to a node in the network by $\mathsf{ID}_{\max}$, that is, $\mathsf{ID}_{\max} := \max_{v \in V} \mathsf{ID}_v$, and by ℓ the node possessing this ID. Note that the subset of natural numbers that can be assigned to the nodes is not restricted to $[n]$; it can be chosen arbitrarily, as long as it contains n distinct IDs. Apart from the IDs, the nodes are identical. Algorithms are *uniform* by default, i.e., the nodes neither know the size of the network n nor any bound on it; for non-uniform algorithms, the nodes may be equipped with such knowledge.

In this paper, we consider rings, i.e., connected graphs where all nodes have degree 2. Neighboring nodes communicate by sending messages to each other. We assume that all messages are subject to corruption: the content of any message is completely erased by noise, resulting in an empty message of length 0, which we call a *pulse*. Computations that ignore any potential content of a message and thus work only with pulses are called *content-oblivious*.

Further, the network is *asynchronous*: the time it takes a pulse to travel through a channel and arrive at its end is unpredictable; the delays are unbounded but always finite. Pulses cannot be dropped or injected by the channel. In such an asynchronous network, the nodes possess neither a common clock nor any notion of time, and they are assumed to be *event-driven*. This means that a node may act once right in the beginning of the computation and from then on only upon receiving a pulse. As a function of its own ID, the previously received pulses, and possibly its own source of randomness, it can then change its state and, for each connected channel, send any number of pulses. Most of our algorithms are deterministic, except for the ones in Section 5, where we consider a special case where nodes do not have IDs and use randomness in order to generate them. We say that an algorithm *terminates* (sometimes also referred to as process termination, explicit termination, or termination detection in the literature) if for each node there is a time at which it has decided on an output and entered a terminating state. Once in a terminating state, a node ignores all incoming pulses and does not send any new ones. We say that an algorithm has *quiescent termination* if at the time the last node terminates it is guaranteed that no pulses are still in transit. An algorithm's *message complexity* is the total number of messages (pulses) it sends during a worst-case computation until all nodes have terminated or until the network has reached quiescence.

Ring's orientation. In a ring, each node communicates with its two neighbors via Port_0 and Port_1. Consider a pulse re-sent from Port_1 by every node receiving it. If such a pulse passes through all edges (i.e., is never reflected by a node with misaligned ports), we call the ring *oriented* and the pulse *clockwise* (CW). Port_1 of each node is then called its CW port, leading to its CW neighbor. Counterclockwise (CCW) is defined analogously via Port_0. Note that CW pulses are sent from CW ports but arrive at CCW ports, and vice versa.

However, rings may not be oriented to begin with. In a *non-oriented* ring, there is no guarantee that Port_0 and Port_1 of a node are aligned with a clockwise or counterclockwise walk on the ring. See Fig. 1 for a demonstration. Algorithms for non-oriented rings must work correctly for all assignments of the nodes' ports. In this case the CW/CCW direction is local per node, and we will use the notion of Port_0 and Port_1 to avoid confusion.

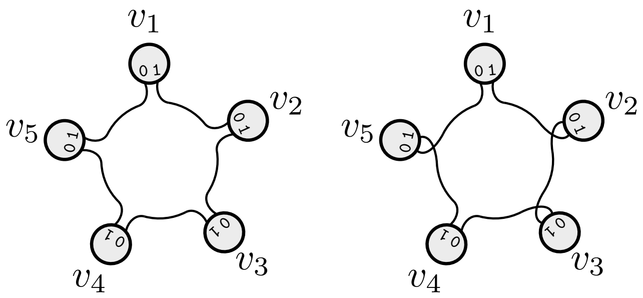


Figure 1 An oriented ring (left) and a non-oriented one (right).

3 Leader Election in an Oriented Ring

In this section, we consider the problem of electing a leader in an oriented ring of size unknown to the nodes. Recall that in the *leader election task*, nodes are required to terminate with an output: Leader or Non-Leader, where a single node $\ell \in V$ outputs Leader, while all other nodes must output Non-Leader. We design an algorithm to elect a leader that features quiescent termination and prove the following.

▶ **Theorem 1.** *There is a* quiescently terminating *content-oblivious algorithm of message complexity* $n(2 \cdot \mathsf{ID}_{\max} + 1)$ *that elects a leader in oriented rings of* n *nodes with unique IDs.*

For the algorithms in this section, we will use the following methods for sending and receiving pulses. The method sendCW() sends one pulse over the CW channel. The method recvCW() checks whether pulses are waiting in the CW incoming queue. If no pulse is in the queue, the method returns 0. Otherwise, it consumes a single pulse from the queue and returns 1. The methods sendCCW() and recvCCW() are the analogous CCW versions. Moreover, for each node v, we introduce counters ρ_{cw} and σ_{cw} for the total number of received and sent CW pulses, respectively. Likewise, we introduce ρ_{ccw} and σ_{ccw} for CCW pulses. We assume each of the four methods above updates those counters with every received or sent pulse. More precisely, every node v has variables $\rho_{\mathsf{cw}}[v]$ and $\sigma_{\mathsf{cw}}[v]$ initially set to 0. Every time recvCW() processes a pulse from the queue of incoming pulses or sendCW() sends a pulse, the counters increase according to $\rho_{\mathsf{cw}}[v] \leftarrow \rho_{\mathsf{cw}}[v] + 1$ and $\sigma_{\mathsf{cw}}[v] \leftarrow \sigma_{\mathsf{cw}}[v] + 1$, respectively.

3.1 Warm-up: Leader Election Without Termination

To demonstrate some of the main ideas of our algorithm, let us begin with a simple quiescently stabilizing algorithm that uses only clockwise pulses and elects a leader in an oriented ring. We emphasize that this algorithm is non-terminating. In this algorithm (see Algorithm 1), each node starts by sending one pulse and then relays every received pulse in the same direction, except for the single time when the number of received pulses reaches its own ID. In this event, the node does not relay this one pulse and assigns itself the state of Leader, at least temporarily. However, any pulses received after this are relayed again, and revert the node to being a Non-Leader.

The main intuition behind this algorithm is that each node will eventually have sent and received exactly $\mathsf{ID}_{\max}$ pulses: n pulses are being generated at the initialization, and these pulses keep circulating in the ring, increasing the counter of received pulses until some node has received as many pulses as its ID. At this point, the node removes one pulse, and we are left with $n-1$ pulses in circulation. Except for the node with $\mathsf{ID}_{\max}$, every node will receive more pulses than its ID, ensuring it eventually becomes a follower. This continues until all nodes, except the one with $\mathsf{ID}_{\max}$, have removed a single pulse from the circulation and declared themselves followers. The last remaining pulse keeps circulating until all nodes have received exactly $\mathsf{ID}_{\max}$ pulses. As soon as the last node receives its $\mathsf{ID}_{\max}$-th pulse, it sets itself as a leader and removes the last remaining pulse. Since the network no longer contains any pulse, which we call quiescence, the states of the event-driven nodes remain unchanged as well. Note, however, that nodes do not terminate since they do not know whether the ring has achieved this quiescent state or not, i.e., whether some pulses are still in transit.

Algorithm 1 Quiescently Stabilizing Leader Election for Node v.

```
sendCW()
while true do
    if recvCW() returns 1 then
        if $\rho_{\mathsf{cw}} = \mathsf{ID}_v$ then
            state ← Leader
        else                                ▷ $v$ acts as a relay unless $\rho_{\mathsf{cw}} = \mathsf{ID}_v$
            state ← Non-Leader
            sendCW()
```

Even though Algorithm 1 does not terminate, analyzing it will help us design our terminating leader election algorithm (Algorithm 2) later on. We begin by stating some key invariants of Algorithm 1, used to show the algorithm's correctness. Their proofs can be found in the full version of the paper [22].

▶ **Lemma 6.** *For every node v running Algorithm 1, the following invariants hold at the end of each iteration of the main loop:*

1. *If $\rho_{\mathsf{cw}} < \mathsf{ID}_v$, then $\sigma_{\mathsf{cw}} = \rho_{\mathsf{cw}} + 1$, i.e., v has sent exactly one pulse more than it has received.*
2. *If $\rho_{\mathsf{cw}} \geq \mathsf{ID}_v$, then $\sigma_{\mathsf{cw}} = \rho_{\mathsf{cw}}$, i.e., v has sent exactly as many pulses as it has received.*

▶ **Lemma 7.** *Let ℓ be the node with $\mathsf{ID}_\ell = \mathsf{ID}_{\max}$. If $\rho_{\mathsf{cw}}[\ell] \geq \mathsf{ID}_\ell$ at some point, then $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ holds for every node v at this point. That is, ℓ is the last node to satisfy $\rho_{\mathsf{cw}}[\ell] \geq \mathsf{ID}_\ell$.*

We now show that quiescence has been reached when $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ holds for all nodes.

▶ **Lemma 8.** *If $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ holds at every node v, then the network is in quiescence.*

Proof. By Lemma 2, we get that $\sigma_{\mathsf{cw}}[v] = \rho_{\mathsf{cw}}[v]$ holds for all v; hence, the total number of sent pulses is equal to the total number of received pulses. In particular, no pulses are in transit (sent but not received). ◀

In fact, the converse is also true; that is, a necessary condition for quiescence is that $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ holds at every node v and becomes a relay.

▶ **Lemma 9.** *If the network is in quiescence, then $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ holds at every node v.*

Proof. If there is quiescence, then $\sum_v \sigma_{\mathsf{cw}}[v] = \sum_v \rho_{\mathsf{cw}}[v]$.

By Lemma 6, every node v has $\sigma_{\mathsf{cw}}[v] \geq \rho_{\mathsf{cw}}[v]$. Assuming there is some bad node b with $\rho_{\mathsf{cw}}[b] < \mathsf{ID}_b$, then by Lemma 1, $\sigma_{\mathsf{cw}}[b] = \rho_{\mathsf{cw}}[b] + 1$. Therefore, $\sum_v \sigma_{\mathsf{cw}}[v] = \sum_{v \neq b} \sigma_{\mathsf{cw}}[v] + \sigma_{\mathsf{cw}}[b] \geq \sum_v \rho_{\mathsf{cw}}[v] + 1$, so this cannot occur. ◀

▶ **Corollary 10.** *There is quiescence at some point of time if and only if each node v has $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ at that point of time.*

Proof. The corollary follows directly from Lemmas 8 and 9. ◀

Another equivalent statement to the ones in Corollary 10 is that every node has sent and received exactly $\mathsf{ID}_{\max}$ pulses.

▶ **Lemma 11.** *In any execution of Algorithm 1, at any point of time, the following statements are equivalent:*

1. *The network is in quiescence,*
2. $\forall v : \rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$*, and*
3. $\forall v : \rho_{\mathsf{cw}}[v] = \sigma_{\mathsf{cw}}[v] = \mathsf{ID}_{\max}$.

Proof. The first two statements are equivalent by Corollary 10.

Now, assuming $\rho_{\mathsf{cw}}[v] = \mathsf{ID}_{\max}$ holds for every v, then so does $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$. Conversely, assume $\forall v : \rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$. By Lemma 7, the node ℓ with the largest ID is the last to satisfy that inequality. After the iteration where this occurs for the first time, we have

$$\rho_{\mathsf{cw}}[\ell] = \mathsf{ID}_\ell = \mathsf{ID}_{\max}.$$

By the first equivalence, there is quiescence, and no more pulses are sent or received. Any pulses that have been sent by a node u have been received by its neighbor v, that is, over all CW edges (u, v), it holds that $\sigma_{\mathsf{cw}}[u] = \rho_{\mathsf{cw}}[v]$.

Also, by Lemma 2, every node v has $\sigma_{\mathsf{cw}}[v] = \rho_{\mathsf{cw}}[v]$. As the network forms a ring, combining those equations yields that $\rho_{\mathsf{cw}}[v] = \sigma_{\mathsf{cw}}[v] = \mathsf{ID}_{\max}$ holds for every v. ◀

Given the above properties, we are ready to prove that certain interesting events occur in every execution of Algorithm 1.

First, we show that the inequality $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ is eventually satisfied by every v. Due to the equivalences given by Lemma 11, this leads to a setting where quiescence is achieved, and every node has sent and received exactly $\mathsf{ID}_{\max}$ pulses.

▶ **Lemma 12.** *In any execution of Algorithm 1, for every node v, there is some iteration, where $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ holds.*

Proof. Let us track the evolution through time of B, the set of nodes that have not met the condition yet; that is, at every point of time, for all $b \in B$, $\rho_{\mathsf{cw}}[b] < \mathsf{ID}_v$.

Initially, B contains all nodes, each of which are removed upon meeting the condition. Since a removed node never enters B again, $|B|$ is monotonically decreasing.

Consider the point of time where $|B|$ reaches a minimum, so B remains fixed. If $|B| = 0$, then there is nothing to prove, so assume $|B| > 0$. At that time, maintain values $\Delta_b := \mathsf{ID}_b - \rho_{\mathsf{cw}}[b] > 0$ for every $b \in B$.

By Lemma 1, for all $b \in B$, we have that $\sigma_{\mathsf{cw}}[b] = \rho_{\mathsf{cw}}[b] + 1$ always holds. Also, from that point on, for every $v \notin B$, we have $\sigma_{\mathsf{cw}}[v] = \rho_{\mathsf{cw}}[v]$ by Lemma 2.

The number of pulses in transit[2] at any given time is the difference between the total number of sent and received pulses across all nodes. Therefore, since $\sum_v \sigma_{\mathsf{cw}}[v] = \sum_v \rho_{\mathsf{cw}}[v] + |B|$, there are still $|B|$ pulses in the network. Since the nodes outside B act as relays, they maintain the number of pulses in transit and, in particular, never remove a pulse from the network. Eventually, some node $b \in B$ must receive a pulse, which decreases the difference Δ_b by one. If $\Delta_b = 0$, then b is removed from B, and $|B|$ decreases beyond its minimum, a contradiction. Otherwise, b forwards a pulse and the number of pulses in transit remains $|B|$, so we can re-apply this argument. At some point, Δ_b reaches 0 for some $b \in B$. Thus, $\rho_{\mathsf{cw}}[b] \geq \mathsf{ID}_b$ holds for b, and b is removed from B, again, a contradiction. ◀

[2] Including pulses that are in some node's queue, but were not processed yet.

A result of Lemma 12, is that, eventually, all nodes send and receive exactly $\mathsf{ID}_{\max}$ pulses, and no further activity occurs on the network.

▶ **Corollary 13.** *In any execution of Algorithm 1, at some point, every node has sent and received exactly* $\mathsf{ID}_{\max}$ *pulses, and the network reached quiescence.*

Proof. By Lemma 12, there is an iteration, after which $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ holds for all nodes v. The statement then follows directly from Lemma 11. ◀

We also have the following trivial upper bound as a direct corollary.

▶ **Corollary 14.** *In any execution of Algorithm 1, for every node* v*, at every point of time* $\rho_{\mathsf{cw}}[v] \leq \mathsf{ID}_{\max}$.

Proof. Immediate from Corollary 13 and the monotonicity of $\rho_{\mathsf{cw}}[v]$, which is initially 0. ◀

3.2 Leader Election With Quiescent Termination

Although Algorithm 1 is a quiescently stabilizing algorithm for leader election, more ideas are needed in order to achieve this with quiescent termination. The main idea is to utilize the CCW channel to notify all nodes when the leader is elected. The immediate approach would be to leverage the event $\rho_{\mathsf{cw}}[v] = \mathsf{ID}_v$, which signifies the successful election once it happens at the node with maximal ID. However, this is impossible, since the same event also occurs for every other node before the election process has finished.

To be able to detect termination, we require an event that occurs *uniquely* for the leader, and never for other nodes. If we managed to run Algorithm 1 over the CCW channel after a full execution of the same algorithm over the CW channel, then, by symmetry, all nodes would eventually receive $\mathsf{ID}_{\max}$ many CCW pulses. In this scenario, the event $\rho_{\mathsf{cw}}[v] = \mathsf{ID}_v = \rho_{\mathsf{ccw}}[v]$ would, in fact, be unique to the node with the largest ID. Indeed, an initial full execution of Algorithm 1 over the CW channel guarantees that all other nodes have $\rho_{\mathsf{cw}}[v] > \mathsf{ID}_v$ prior to exchanging CCW pulses, so the above event occurs uniquely at the leader and could be used as the trigger for termination.

The main remaining difficulty is that we *cannot* start the CCW algorithm after the CW one is done since neither the leader nor any other node can infer that the CW algorithm has stabilized purely from the number of CW pulses received. We overcome this obstacle by running both algorithms in parallel, ensuring that the CCW one lags behind the CW one. By subtly prioritizing the execution of the CW algorithm over that of the CCW one, we enforce that once $\rho_{\mathsf{ccw}}[v] = \mathsf{ID}_v$ occurs for some Non-Leader node, we already have $\rho_{\mathsf{cw}}[v] > \mathsf{ID}_v$. Then, the only node v satisfying $\rho_{\mathsf{cw}}[v] = \mathsf{ID}_v = \rho_{\mathsf{ccw}}[v]$ is the elected leader. It is the uniqueness of all IDs, crucially including $\mathsf{ID}_{\max}$, that enables this approach.

The complete algorithm with quiescent termination is given in Algorithm 2. It contains two instances of Algorithm 1: one over the CW channel (lines 3–8) and one over the CCW channel (lines 9–13). The CW instance starts, as before, upon initialization, while the CCW instance starts at node v once it reaches the $\rho_{\mathsf{cw}}[v] = \mathsf{ID}_v$ event in the CW instance. This guarantees that the CCW instance lags behind the CW one. Finally, the last part of the algorithm (lines 14–17) is executed once the condition $\rho_{\mathsf{cw}}[v] = \mathsf{ID}_v = \rho_{\mathsf{ccw}}[v]$ is observed to be true, which happens only for the leader. At this point, all the nodes have ρ_{cw} and ρ_{ccw} set to $\mathsf{ID}_{\max}$, and the leader was the last node for which this event occurred, triggering the following termination process: the leader sends a single CCW pulse. Any node receiving this extra pulse sees, for the first time, $\rho_{\mathsf{ccw}} > \rho_{\mathsf{cw}}$, forwards the pulse and terminates (Algorithm 2). The extra pulse is forwarded until it returns to the leader, causing it to terminate without forwarding the pulse. The analysis of Algorithm 2 is deferred to the full version of the paper [22] due to the space constraints.

Algorithm 2 Quiescently Terminating Leader Election for Node v.

```
sendCW()
repeat
    if recvCW() returns 1 then                          ▷ Run Algorithm 1 over the CW channel
        if $\rho_{\mathsf{cw}} = \mathsf{ID}_v$ then
            state ← Leader
        else
            state ← Non-Leader
            sendCW()
    if $\rho_{\mathsf{cw}} \geq \mathsf{ID}_v$ then     ▷ Run Algorithm 1 over the CCW channel, once $\rho_{\mathsf{cw}} \geq \mathsf{ID}_v$
        if $\sigma_{\mathsf{ccw}} = 0$ then sendCCW() end if
        if recvCCW() returns 1 then
            if $\rho_{\mathsf{ccw}} \neq \mathsf{ID}_v$ then
                sendCCW()
    if $\rho_{\mathsf{cw}} = \mathsf{ID}_v = \rho_{\mathsf{ccw}}$ then     ▷ Initiate a termination pulse
        sendCCW()
        while recvCCW() returns 0 do
            pass                                        ▷ Wait for return of termination pulse
until $\rho_{\mathsf{ccw}} > \rho_{\mathsf{cw}}$
output state
```

4 Leader Election in Non-Oriented Rings

A natural follow-up question to the above leader election algorithm in oriented rings is whether the same holds for *non-oriented* rings. In this setting, nodes do not possess a predefined CW channel and CCW channel anymore. Instead, they have two ports, Port_0 and Port_1, connecting them to their two neighbors in an arbitrary order.

The straightforward approach would be to first orient the ring with a quiescently terminating algorithm and then use our leader election algorithm from Section 3. Since orienting the ring is easy with leader elected by quiescent termination, orientation and leader election are essentially equivalent tasks from the perspective of quiescently terminating algorithms.

In this section, we instead present a quiescently *stabilizing* algorithm for non-oriented rings, which both elects a leader and orients the ring. Recall that the difference from quiescent termination is that the nodes do not need to terminate explicitly; it suffices for all pulse activity to cease with the correctly computed output still present. Recall our main theorem for this part.

▶ **Theorem 2.** *There is a quiescently stabilizing content-oblivious algorithm of message complexity $n(2 \cdot \mathsf{ID}_{\max} + 1)$ that elects a leader and orients a non-oriented ring of n nodes with unique IDs.*

We emphasize again that this algorithm does not terminate in the usual sense. Instead, its success is defined as reaching quiescence while guaranteeing that at that time only a single node has set its internal state to Leader, while all other nodes have set their state to Non-Leader. Additionally, we require that nodes achieve a consistent orientation of the ring as follows: each node has to label exactly one of its two ports as the port leading to the CW neighbor such that starting at some node and repeatedly moving to node connected to CW port lets us pass through all edges in the ring.

For ease of exposure, we first present an algorithm of slightly worse complexity but whose analysis is almost fully captured by results from earlier sections. We then improve its complexity by proving an additional property about Algorithm 1, our main building block for our algorithm for non-oriented rings (Algorithm 3). Namely, we show that executing Algorithm 1 on a ring with non-unique IDs essentially achieves the same guarantees as when used on a ring with unique IDs (Lemma 16). This observation allows us to halve the complexity of Algorithm 3, as we shall see, and also has implications for solving the same tasks on anonymous rings with access to randomness. As this extension to anonymous rings is a minor effort and follows from standard techniques, we defer it to Section 5.

Algorithm overview. At a high level, Algorithm 3 consists of two parallels executions of Algorithm 1, each one using each channel in the ring in a single direction. For an intuition of how that is possible, consider a setting where the network has a single pulse in transit. Suppose that all nodes execute the same algorithm that sends a pulse on Port_1 whenever one is received on Port_0, and vice versa. Forwarding the pulse in this manner has it travel the entire ring, since every time a pulse is received by a given node v from one of its neighbors, it is sent to its other neighbor. If adding another pulse going in the other direction to the network, the two pulses independently travel around the ring in opposite directions. The nodes can thus effectively run two algorithms in parallel without them interfering with one another, as long as one only works with clockwise pulses and the other with counterclockwise pulses. As Algorithm 1 only uses pulses going in one direction, it satisfies this requirement. The major caveat is that the nodes cannot be certain which of the two algorithms they execute is working with clockwise pulses and which is working with counterclockwise pulses.

Our algorithm has essentially two parts: one in which the node reacts to pulses and possibly forwards them (Lines 5 to 7), and one in which it computes its output depending on the number of pulses it received from each port (Lines 8 to 16). From the point of view of analyzing how many pulses are eventually sent in the network, only the first part is relevant. It is the part that simulates two executions of Algorithm 1. For the nodes to settle on a consistent ring orientation in the second part, we break symmetry between the two options by having strictly more pulses sent in one orientation of the ring than in the other.

We distinguish the two parallel executions of Algorithm 1 by having each node v pick two distinct virtual IDs for itself, $\mathsf{ID}_v^{(0)}$ and $\mathsf{ID}_v^{(1)}$ (Algorithm 3). $\mathsf{ID}_v^{(1)}$ affects how v behaves regarding pulses received from its Port_0, and symmetrically for $\mathsf{ID}_v^{(0)}$ and Port_1. While nodes do not know which of their two virtual IDs is used in the clockwise or counterclockwise execution of Algorithm 1, they use a distinct ID in both executions. Eventually, in each direction, the number of pulses received by each node stabilizes to the largest ID in that direction. The choice of IDs ensures that the two parallel executions have distinct largest IDs, so eventually all nodes see strictly more pulses being sent in one direction than the other. This allows nodes to agree on a common orientation of the ring, and elect as leader the node who was the source of the largest ID.

The nodes use the following methods for sending and receiving pulses. Let $i \in \{0, 1\}$.

1. $\text{sendPort}_i()$: sends a pulse through Port_i,
2. $\text{recvPort}_i()$: check whether a pulse is waiting in the incoming queue of Port_i. If not, return 0. Otherwise, consume a single pulse from the queue and return 1.

▶ **Proposition 15.** *Algorithm 3 elects a leader and consistently orients the ring using $n(4\mathsf{ID}_{\max} - 1)$ pulses. It achieves quiescence but does not terminate.*

Algorithm 3 Quiescently Stabilizing Leader Election on Non-Oriented Rings for Node v.

```
for $i \in \{0,1\}$ do
    $\mathsf{ID}_v^{(i)} \leftarrow 2 \cdot \mathsf{ID}_v - 1 + i$
    sendPort$_i$()
while true do
    for $i \in \{0,1\}$ do
        if recvPort$_{1-i}$() returns 1 and $\rho_{1-i} \neq \mathsf{ID}_v^{(i)}$ then
            sendPort$_i$() ▷ Pulses received at one port are sent forward at the opposite one
    if $\max(\rho_0, \rho_1) \geq \mathsf{ID}_v^{(1)}$ then
        if $\rho_0 = \mathsf{ID}_v^{(1)}$ and $\rho_1 < \mathsf{ID}_v^{(1)}$ then
            state ← Leader
        else
            state ← Non-Leader
        if $\rho_0 > \rho_1$ then
            name Port$_0$ := CCW and Port$_1$ := CW     ▷ Port$_0$ connects the CCW neighbor
        else
            name Port$_0$ := CW and Port$_1$ := CCW
```

Proof. Consider ℓ, the node of largest ID. We define as clockwise the direction of a pulse sent from ℓ's Port$_1$ and forwarded by all other nodes (i.e., sent from Port$_i$ after arriving at Port$_{1-i}$). We call the ports sending such a pulse clockwise, and those receiving it counterclockwise. We show the our algorithm elects ℓ as a leader and all nodes declare the correct port as clockwise.

We show that the network eventually achieves quiescence and all nodes receive the same number of clockwise and counterclockwise pulses. Let us argue this for clockwise pulses; the property for counterclockwise pulses will follow by symmetry. For every node v whose Port$_1$ connects it to its clockwise neighbor, let us rename sendPort$_1$() to sendCW() and recvPort$_0$() to recvCW() in its code. Let us also define $\mathsf{ID}_v^{\text{cw}} = \mathsf{ID}_v^1$ for such nodes. For the other nodes, which are connected to their clockwise neighbors through Port$_0$, rename sendPort$_0$() to sendCW() and recvPort$_1$() to recvCW() in their code, and let $\mathsf{ID}_v^{\text{cw}} = \mathsf{ID}_v^0$. We emphasize that this renaming is done purely for our analysis and is not an operation performed by the nodes, which are oblivious to what the clockwise direction is. Consider an execution of Algorithm 3. Whenever the scheduler delivers a clockwise pulse, this pulse is read by a recvCW() method, and if forwarded (which depends on $\mathsf{ID}_v^{\text{cw}}$), it is re-sent by a sendCW() method, according to our renaming. Other methods are never activated by clockwise pulses and never emit a clockwise pulse. As such, Algorithm 3 executes the exact same code on clockwise pulses as Algorithm 1 and thus has the same guarantees as Algorithm 1 regarding clockwise pulses. By Corollary 13, this means that we achieve quiescence for clockwise pulses, with all nodes eventually having sent and received exactly the same number $\max_v \mathsf{ID}_v^{\text{cw}}$ of clockwise pulses. The same holds symmetrically for $\max_v \mathsf{ID}_v^{\text{ccw}}$ pulses counterclockwise pulses per node, where $\mathsf{ID}_v^{\text{ccw}}$ is defined similarly to $\mathsf{ID}_v^{\text{cw}}$.

Since ℓ picks as identifiers $2 \cdot \mathsf{ID}_{\max}$ and $2 \cdot \mathsf{ID}_{\max} - 1$ for the two directions, we have $\max_v \mathsf{ID}_v^{\text{cw}} = 2 \cdot \mathsf{ID}_{\max}$ and $\max_v \mathsf{ID}_v^{\text{ccw}} = 2 \cdot \mathsf{ID}_{\max} - 1$. Hence, all nodes have sent and received $2 \cdot \mathsf{ID}_{\max}$ clockwise pulses and $2 \cdot \mathsf{ID}_{\max} - 1$ counterclockwise pulses. This yields a bound of $n(4 \cdot \mathsf{ID}_{\max} - 1)$ pulses, ensuring a consistent orientation according to the test on Algorithm 3. ◀

Improving the message complexity. We now show how to improve the complexity of Algorithm 3 to $n(2 \cdot \mathsf{ID}_{\max} + 1)$. Since nodes send pulses according to their IDs, Algorithm 3 effectively doubles the number of pulses sent by the algorithm. To avoid this doubling, one can generate the two IDs in a different manner; for instance, $\mathsf{ID}_v^{(1)} \leftarrow \mathsf{ID}_v + 1$, and $\mathsf{ID}_v^{(0)} \leftarrow \mathsf{ID}_v$. However, this leads to assigning the same ID to multiple nodes. We argue that Algorithm 3 works correctly even when IDs are not unique as long as the largest clockwise and counterclockwise IDs are different. To that goal, we need to re-analyze Algorithm 1 in such case, which we do in the next two technical lemmas.

▶ **Lemma 16.** *Corollary 13 still holds if nodes run Algorithm 1 with non-unique IDs. This includes the case in which multiple nodes v have $\mathsf{ID}_v = \mathsf{ID}_{\max}$.*

Note that Lemma 16 is about Algorithm 1, the implicit main subroutine of Algorithm 3, not Algorithm 3 itself. Most elements of the proof of Corollary 13 make no reference to the node of largest ID. Most importantly, the invariants of Lemma 6 (that $\sigma_{\mathsf{cw}}[v] = \rho_{\mathsf{cw}}[v] + 1$ while $\rho_{\mathsf{cw}}[v] < \mathsf{ID}_v$, and $\sigma_{\mathsf{cw}}[v] = \rho_{\mathsf{cw}}[v]$ once $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$) are consequences of how each node reacts to the pulses it receives, and changing the distribution of IDs does not change that. Lemma 7, however, makes an explicit reference to a node of largest ID, which requires us to change the argument somewhat. Lemma 17, which we shall now prove, generalizes Lemma 7 to the setting with non-unique IDs.

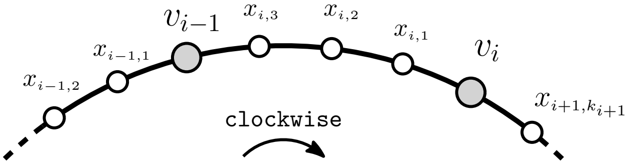


Figure 2 The naming of nodes between nodes of largest ID in the proof of Lemma 17.

▶ **Lemma 17.** *Consider the set of nodes of largest ID, $V_{\max} = \{v : \mathsf{ID}_v = \mathsf{ID}_{\max}\}$. In Algorithm 1, if $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ for all $v \in V_{\max}$ at some point, then $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ holds for every node v at this point. That is, one of the nodes in $V_{\max}$ is the last node to satisfy $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$.*

Proof. Let $m = |V_{\max}|$ be the number of nodes that hold $\mathsf{ID}_{\max}$. Denote these nodes $v_1, \ldots, v_m$, ordered clockwise from an arbitrary one of them. Let us identify $v_{m+1} = v_1$ for ease of notation. For each $i \in [m]$, let $k_i \in \{0, \ldots, n-1\}$ be the number of consecutive nodes with $\mathsf{ID}_v < \mathsf{ID}_{\max}$ preceding v_i in the ring. For each $j \in [k_i]$, let $x_{i,j}$ be the node j counterclockwise hops from v_i in the ring. See Figure 2 for an illustration.

We show that if $\rho_{\mathsf{cw}}[v_i] \geq \mathsf{ID}_{v_i}$ holds at v_i, then it also holds at all $x_{i,j}$, $j \in [k_i]$. Since every node $v \notin V_{\max}$ has a node of largest ID in its clockwise direction later in the ring, this implies that when $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ holds at all $v_i \in V_{\max}$, it also holds at every node $v \in V$.

Consider $v_i \in V_{\max}$ s.t. $\rho_{\mathsf{cw}}[v_i] \geq \mathsf{ID}_{v_i} = \mathsf{ID}_{\max}$. Suppose $k_i > 0$, as otherwise the result is trivial. Let $x_{i,0} = v_i$. For each $j \in [0, k_i)$, we show that $\rho_{\mathsf{cw}}[x_{i,j}] \geq \mathsf{ID}_{\max}$ implies $\rho_{\mathsf{cw}}[x_{i,j+1}] \geq \mathsf{ID}_{\max}$. As the base case $\rho_{\mathsf{cw}}[x_{i,0}] = \rho_{\mathsf{cw}}[v_i] \geq \mathsf{ID}_{\max}$ holds, we get our result by induction. Let $j \in [0, k_i)$ and suppose $\rho_{\mathsf{cw}}[x_{i,j}] \geq \mathsf{ID}_{\max}$. Since $\rho_{\mathsf{cw}}[x_{i,j}] \leq \sigma_{\mathsf{cw}}[x_{i,j+1}]$, we have that $\sigma_{\mathsf{cw}}[x_{i,j+1}] \geq \mathsf{ID}_{\max}$. Since $\mathsf{ID}_{\max} > \mathsf{ID}_{x_{i,j+1}}$, by Lemma 6, it needs to hold that $\rho_{\mathsf{cw}}[x_{i,j+1}] = \sigma_{\mathsf{cw}}[x_{i,j+1}]$. Therefore, $\rho_{\mathsf{cw}}[x_{i,j+1}] \geq \mathsf{ID}_{\max}$. ◀

Equipped with Lemma 17, the proof of Lemma 16 follows quite naturally.

Proof of Lemma 16. Let us review the proof of Corollary 13, and see which elements of it could be affected by multiple nodes having the same ID. Let $V_{\max} = \{v : \mathsf{ID}_v = \mathsf{ID}_{\max}\}$ be the set of nodes of largest ID.

As already stated, the invariants of Lemma 6 still hold, and arguments relying on Lemma 7 must be amended to rely on Lemma 17 instead. Lemmas 8 and 9 and Corollary 10, about how the network is in quiescence if and only if each node v has received (and, by Lemma 6, also sent) at least ID_v pulses, are immediate consequences of Lemma 6, and hence still hold.

We now get to Lemmas 11 and 12, which are two lemmas from which Corollary 13 follows most directly. Lemma 11 shows equivalences between three properties: quiescence, that $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ at each v, and that $\rho_{\mathsf{cw}}[v] = \sigma_{\mathsf{cw}}[v] = \mathsf{ID}_{\max}$ at each v. Again, the arguments still hold if IDs are not unique: the connection between quiescence and all nodes satisfying $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$ was shown in Corollary 10; while some node $v \in V_{\max}$ has received less than $\mathsf{ID}_{\max}$ pulses, the network is not in quiescence; for all nodes in $V_{\max}$ to have received $\mathsf{ID}_{\max}$ pulses, other nodes in the network need to have sent this many pulses, which implies they have also received $\mathsf{ID}_{\max}$ pulses.

Lemma 12 shows that the network cannot permanently remain in a state in which some nodes have $\rho_{\mathsf{cw}}[v] < \mathsf{ID}_v$. The argument again does not rely on the uniqueness of the IDs, and only uses that if some non-empty set B of nodes satisfies $\rho_{\mathsf{cw}}[v] < \mathsf{ID}_v$ for each $v \in B$, then since for those nodes $\sigma_{\mathsf{cw}}[v] = \rho_{\mathsf{cw}}[v] + 1$ (Lemma 6) the network has pulses still in transit. These pulses must eventually reach nodes in B, increase the number of received pulses ρ_{cw} of nodes in it, and eventually to the point that for a node in B, $\rho_{\mathsf{cw}}[v] \geq \mathsf{ID}_v$.

Put together, the whole argument still holds with non-unique IDs. ◀

The proof of our second main theorem is a corollary of the above.

Proof of Theorem 2. Let us modify Algorithm 3 as follows:

- In Algorithm 3, we set as IDs $\mathsf{ID}_v^{(1)} \leftarrow \mathsf{ID}_v + 1$, and $\mathsf{ID}_v^{(0)} \leftarrow \mathsf{ID}_v$.

Similar to the argument in the proof of Proposition 15, this amounts to running two parallel instances of Algorithm 1 over each channel. As in that proof, consider the maximal clockwise and counterclockwise IDs, $\max_v \mathsf{ID}_v^{\mathrm{cw}} = \mathsf{ID}_{\max} + 1$ and $\max_v \mathsf{ID}_v^{\mathrm{ccw}} = \mathsf{ID}_{\max}$. From Lemma 16, we know that the number of clockwise pulses sent and received by each node eventually stabilizes at $\mathsf{ID}_{\max} + 1$, and similarly stabilizes at $\mathsf{ID}_{\max}$ for the counterclockwise direction. This results in a single leader being elected and a consistent orientation as before. The upper bound of $n(2\mathsf{ID}_{\max} + 1)$ follows from $n(\mathsf{ID}_{\max} + 1)$ pulses being exchanged in one direction, and $n \cdot \mathsf{ID}_{\max}$ in the other. ◀

5 Anonymous Rings

In this section, we consider the setting where a ring consists of n identical nodes without IDs, each with access to its own independent source of randomness. We call such a ring *anonymous*. As is standard in the literature about randomized algorithms, we aim to solve our computational task *with high probability*, defined as bounding the probability of failure by an arbitrary negative power of the network size n. That is, we present an algorithm parameterized by a value $c > 0$, such that for any n, the algorithm correctly elects a leader in rings of size n with probability at least $1 - O(n^{-c})$. Our algorithm also correctly orients a non-oriented ring with high probability.

Similar to other sections of this paper, proofs omitted from this section can be found in the full version of this paper [22].

As alluded to in the introduction, electing a leader with quiescent termination is impossible under the assumptions of this section, even if we relax the objective to succeed with only some arbitrary small constant probability. This follows from a negative result by Itai and Rodeh [26, Thm. 4.1] about counting the number of nodes in an anonymous ring because computing this number is easy in a ring with an elected leader. Consequently, we only aim for a quiescently stabilizing algorithm.

As in Section 4, observe that despite assuming the uniqueness of all IDs in Section 3.1 to elect the node with the largest ID, the task remains well-defined even if only the largest ID is unique. Lemma 16 showed that on a ring with possibly non-unique IDs, Algorithm 1 elects as leader all nodes v such that $\mathsf{ID}_v = \mathsf{ID}_{\max}$. Hence, if a single node satisfies $\mathsf{ID}_v = \mathsf{ID}_{\max}$, a single leader is elected. Algorithm 3 is simply two parallel executions of Algorithm 1. As we already showed in the proof of Theorem 2, those two parallel executions still yield the desired result as long as the maximal ID in the two executions is unique. Hence, providing an algorithm for sampling IDs with the guarantee that the maximal ID is unique with high probability is sufficient to obtain a variant of Theorem 2 for anonymous rings.

We outline in Algorithm 4 a process by which nodes can utilize their access to randomness to sample random IDs with the guarantee that, with high probability, the maximal ID is unique and of order $n^{O(1)}$. The algorithm terminates quiescently – in fact, it uses no communication – enabling composition: any algorithm can be performed afterwards with the sampled IDs. As a result, with high probability, the anonymous setting reduces to the one considered in Lemma 16. As explained previously, such an algorithm for sampling IDs immediately implies the following result.

▶ **Theorem 3.** *There is a content-oblivious algorithm of complexity $n^{O(1)}$ that elects a leader and orients an anonymous ring of n nodes, each with access to its own source of randomness, with high probability. The algorithm reaches quiescence but does not terminate.*

At a high level, the algorithm for sampling IDs has each node first sample the number of bits in its ID from a geometric distribution with parameter $p = 2^{-1/\Theta(c)}$ before sampling said bits uniformly at random. While each ID is of expected length $\Theta(c)$ and value $2^{\Theta(c)}$ at the end of this process, the maximal ID is of length $\Theta(c^2 \log n)$ and value $n^{\Theta(c^2)}$, with high probability. See [6]. Intuitively, while sampling a large ID is an unlikely event, this unlikely event becomes more and more likely to occur somewhere in the network as the network grows. Similar ideas have previously appeared in the literature, e.g., in [1, 18].

Algorithm 4 Message-Free Algorithm for Sampling an ID of Order $n^{O(c^2)}$ for Node v, $c > 0$.

```
1: $p \leftarrow 2^{-1/(c+2)}$
2: Sample $BitCount \sim Geo(1-p)$, i.e., according to the geometric distribution with
   parameter $1-p$
3: Sample $\mathsf{ID}_v$ uniformly at random from $\{0,1\}^{BitCount}$
```

▶ **Lemma 18.** *For any constant $c > 0$, with high probability, running Algorithm 4 in an anonymous ring of n nodes assigns each one an ID of size $n^{O(c^2)}$ such that the maximal ID is attained uniquely by one node and is at least $n^{\Omega(c)}$.*

Proof sketch. Each node has a probability of p^x to have their *BitCount* variable exceed some value x. With $x \in \Theta(\log_p n)$, this probability is of order $1/\operatorname{poly}(n)$. Doing a union bound over all nodes, we get that with high probability, all nodes' *BitCount* variables stay below some value $x \in \Theta(\log_p n)$. It follows that the largest ID is of order at most $\operatorname{poly}(n)$. Since nodes do their sampling independently, the probability that all nodes' *BitCount* variables stay below some value x is $(1-p^x)^n$. For $x \in \Theta(\log_p n)$, this is $1/\operatorname{poly}(n)$. As a result, at least one node's *BitCount* variable exceeds $\Theta(\log_p n)$, with high probability. Such nodes then sample their random bits uniformly at random from $\operatorname{poly}(n)$ choices, which makes a collision at the highest value very unlikely. ◀

The proof of Theorem 3 follows immediately from Lemma 18.

Finally, we note that a slight modification of Algorithm 3 would also allow us to sample a unique ID for each node (Proposition 19), with high probability. As a result, the three settings of (1) the anonymous ring, (2) the ring with a leader, and (3) the ring with unique IDs and an orientation, are crucially separated by the possibility of quiescent termination. Leader election, orienting the ring, and assigning unique IDs can all be computed in setting (1). However, they can only be done without termination, while in settings (2) and (3), the same tasks and more can be performed with quiescent termination.

▶ **Proposition 19.** *Let ID_v be the ID sampled by nodes in Algorithm 4, before running Algorithm 3. Modify Algorithm 3 so that whenever a node receives a pulse, if $\min(\rho_0, \rho_1) > \mathsf{ID}_v$, node v updates its ID to a new ID sampled uniformly at random between 1 and $\min(\rho_0, \rho_1) - 1$. Then, with high probability, all nodes have distinct IDs when reaching quiescence.*

6 Lower Bound on Message Complexity in Content-Oblivious Rings

In this section, we show that the dependency of the message complexity of our algorithms on $\mathsf{ID}_{\max}$ is natural and inevitable by providing a lower bound showing that the number of pulses sent increases indefinitely with the number of available IDs.

▶ **Theorem 20.** *Let k and n be arbitrary positive integers, $k \geq n$. If k distinct IDs are assignable to the n nodes of the ring, at least $n\lfloor \log(k/n) \rfloor$ pulses are sent by any leader election algorithm for some assignment of IDs. In particular, an unbounded number of pulses is sent for an infinite supply of IDs even on rings with just a single node.*

The proof of this theorem makes use of the following definition.

▶ **Definition 21** (Solitude pattern)**.** *Consider a ring with a single node ($n = 1$), and fix a specific algorithm. Assume a scheduler that delivers pulses one by one, keeping the order in which they were sent (breaking ties by prioritizing* CW *pulses). Define the* solitude pattern *as the sequence of incoming pulses observed by the node, encoded as a binary string where 0 and 1 encode* CW *and* CCW *pulses, respectively. We denote the solitude pattern of a node with $\mathsf{ID} = i$ by p_i.*

Besides this crucial definition, we make use of the following lemma telling us that each ID has its own, unique solitude pattern. Essentially, the proof relies on matching all possible pairs of IDs against each other in a ring of two nodes. If any two IDs had the same solitude pattern, they would send and receive pulses in this ring ($n = 2$) exactly as they would in solitude ($n = 1$). Thus, in one of these execution they give an invalid output.

▶ **Lemma 22.** *For any uniform content-oblivious leader election algorithm, each solitude pattern is unique. In other words, for any pair of distinct IDs $i \neq j$, we have $p_i \neq p_j$.*

Proof. Fix a content-oblivious leader election algorithm on uniform rings. Seeking contradiction, assume that two nodes with distinct IDs, $i \neq j$, have the same solitude pattern, $p_i = p_j$. Note that each of these nodes outputs Leader when run in isolation ($n = 1$).

Consider a ring with $n = 2$ nodes, which are assigned the IDs i and j, respectively. Assume a scheduler behaving the same way as in the definition of a solitude pattern for each node individually. That is, pulses arrive one by one in the order they were sent out. Moreover, we assume that the same delay is applied to all pulses. Since the two nodes' solitude patterns are identical, and the scheduler maintains order, each node will receive (and thus generate) exactly its solitude pattern. Indeed, since $p_i = p_j$, both nodes send their first pulse in the same direction, and hence both receive it from the same direction as they would when alone and then send their next pulse accordingly; thus both receive and send exactly the pattern $p_i = p_j$. This means that both nodes output Leader as they must in their solitude situation, contradicting the guarantees of the leader election task. ◀

Having established that each ID has its own unique solitude pattern, a lower bound arises from properties of binary strings, namely the length required to avoid repeating patterns. We begin with the following simple property following from the pigeon-hole principle.

▶ **Lemma 23.** *For any $s, n \geq 1$, any set of $n2^s$ distinct binary strings contains n strings sharing a common prefix of length at least s.*

Proof. Let s and n be positive integers. There are only $2^s - 1$ distinct binary strings shorter than s. Therefore, in a set of $n2^s$ distinct binary strings at least $n2^s - 2^s - 1 = (n-1)2^s + 1$ of them have length at least s, implying that they contain a prefix string of length s. There are only 2^s distinct binary prefixes of length s, thus the pigeon-hole principle implies that at least one prefix of length s is shared by at least $\lceil ((n-1)2^s + 1)/2^s \rceil = \lceil n - 1 + 2^{-s} \rceil = n$ of the strings in the set. ◀

After deducing another corollary, we are ready to prove the lower bound of Theorem 20.

▶ **Corollary 24.** *For any integers $k \geq n \geq 1$, any set of k distinct binary strings contains n strings sharing a common prefix of length at least $\lfloor \log(k/n) \rfloor$.*

Proof. Apply Lemma 23 with the given n and $s = \lfloor \log(k/n) \rfloor$. Note that the given set is large enough since $n2^s \leq n2^{\log(k/n)} = n(k/n) = k$. ◀

Proof of Theorem 20. Assume that we have a uniform leader election algorithm, a ring with n nodes, and at least k assignable IDs for any positive integers $k \geq n$. Due to Lemma 22 we know that each ID has its own unique solitude pattern, which is just a binary string. By Corollary 24, there are at least n IDs whose solitude patterns share a common prefix of length $s = \lfloor \log(k/n) \rfloor$. Assume a scheduler that behaves as described in Lemma 22, for all n nodes. It follows that all nodes send and receive pulses in exactly the same way as in their respective solitude situations for the first $\lfloor \log(k/n) \rfloor$ time steps, sending one pulse in each time step. Consequently, at least $n \cdot s = n\lfloor \log(k/n) \rfloor$ pulses are sent in total, proving the first part of the theorem. The last statement in the theorem follows for an infinite set of assignable IDs because $n\lfloor \log(k/n) \rfloor$ grows indefinitely with increasing k, even for $n = 1$. ◀

Theorem 4 is then an immediate corollary of the above since the number of distinct IDs is bounded by $\mathsf{ID}_{\max}$. This lower bound complements the upper bound of Theorem 1 and proves that the $\mathsf{ID}_{\max}$ term is not an artifact of our analysis or algorithm design but, rather, an inherent property of the problem in this setting.

7 Conclusion and Open Questions

As our main result, we have presented a quiescently terminating algorithm for leader election in oriented rings with unique IDs that communicates $n(2 \cdot \mathsf{ID}_{\max} + 1)$ pulses. This implies that any content-oblivious computation can be performed on rings without assuming a pre-existing leader. We have also provided a lower bound showing that the message complexity depending on $\mathsf{ID}_{\max}$ is not a fluke but inherent to the problem.

An immediate candidate for future work is to extend our results from rings to general networks, i.e., to design a content-oblivious leader election algorithm in arbitrary 2-edge connected networks or, alternatively, prove this task impossible. Considering non-oriented rings may be useful towards that goal since there is no sense of direction in general networks. Our content-oblivious leader election for non-oriented rings does not terminate, and we conjecture that this is inherent to the model. It remains as an open task for future work to prove this or find a terminating algorithm.

References

1 Y. Afek and Y. Matias. Elections in anonymous networks. *Information and Computation*, 113(2):312–330, 1994. doi:10.1006/inco.1994.1075.

2 Dana Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, STOC '80, pages 82–93, New York, NY, USA, 1980. ACM. doi:10.1145/800141.804655.

3 Yagel Ashkenazi, Ran Gelles, and Amir Leshem. Noisy beeping networks. *Information and Computation*, 289(A):104925, 2022. doi:10.1016/J.IC.2022.104925.

4 Hagit Attiya and Marc Snir. Better computing on the anonymous ring. *Journal of Algorithms*, 12(2):204–238, 1991. doi:10.1016/0196-6774(91)90002-G.

5 Hagit Attiya, Marc Snir, and Manfred K. Warmuth. Computing on an anonymous ring. *J. ACM*, 35(4):845–875, October 1988. doi:10.1145/48014.48247.

6 F Thomas Bruss and Colm Art O'cinneide. On the maximum and its uniqueness for geometric random samples. *Journal of applied probability*, 27(3):598–610, 1990.

7 James E. Burns. A formal model for message passing systems. Technical report, Computer Science Dept., Indiana Univ., Bloomington, Ind., 1980. TR91.

8 Keren Censor-Hillel, Shir Cohen, Ran Gelles, and Gal Sela. Distributed computations in fully-defective networks. *Distributed Computing*, 36(4):501–528, 2023. doi:10.1007/s00446-023-00452-2.

9 Keren Censor-Hillel, Ran Gelles, and Bernhard Haeupler. Making asynchronous distributed computations robust to noise. *Distributed Computing*, 32(5):405–421, October 2019. doi:10.1007/s00446-018-0343-5.

10 Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979. doi:10.1145/359104.359108.

11 Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In *Distributed Computing, 24th International Symposium, DISC 2010*, volume 6343 of *LNCS*, pages 148–162, 2010, Cambridge, MA, USA, September 13–15, 2010. Proceedings, 2010. Springer. doi:10.1007/978-3-642-15763-9_15.

12 Artur Czumaj and Peter Davies. Leader election in multi-hop radio networks. *Theoretical Computer Science*, 792:2–11, 2019. doi:10.1016/J.TCS.2019.02.027.

13 Peter Davies. Optimal message-passing with noisy beeps. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC*, pages 300–309, 2023, Orlando, FL, USA, June 19–23, 2023, 2023. ACM. doi:10.1145/3583668.3594594.

14 Danny Dolev, Maria Klawe, and Michael Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3):245–260, 1982. doi:10.1016/0196-6774(82)90023-2.

15 Fabien Dufoulon, Janna Burman, and Joffroy Beauquier. Beeping a deterministic time-optimal leader election. In *32nd International Symposium on Distributed Computing, DISC 2018*, volume 121 of *LIPIcs*, pages 20:1–20:17, New Orleans, LA, USA, October 15–19, 2018, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.DISC.2018.20.

16 Fabien Dufoulon, Janna Burman, and Joffroy Beauquier. Can uncoordinated beeps tell stories? In *PODC '20: ACM Symposium on Principles of Distributed Computing*, pages 408–417, Virtual Event, Italy, August 3–7, 2020, 2020. ACM. doi:10.1145/3382734.3405699.

17 M. El-Ruby, J. Kenevan, R. Carlson, and K. Khalil. A linear algorithm for election in ring configuration networks. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, volume i, pages 117–123 vol.1, Los Alamitos, CA, USA, 1991. IEEE Computer Society. doi:10.1109/HICSS.1991.183877.

18 Michael Feldmann, Andreas Padalkin, Christian Scheideler, and Shlomi Dolev. Coordinating amoebots via reconfigurable circuits. *Journal of Computational Biology*, 29(4):317–343, 2022. doi:10.1089/cmb.2021.0363.

19 Paola Flocchini, Evangelos Kranakis, Danny Krizanc, Flaminia L. Luccio, and Nicola Santoro. Sorting and election in anonymous asynchronous rings. *Journal of Parallel and Distributed Computing*, 64(2):254–265, 2004. doi:10.1016/j.jpdc.2003.11.007.

20 Klaus-Tycho Förster, Jochen Seidel, and Roger Wattenhofer. Deterministic leader election in multi-hop beeping networks – (extended abstract). In *Distributed Computing – 28th International Symposium, DISC'14*, volume 8784 of *LNCS*, pages 212–226, 2014, Austin, TX, USA, October 12–15, 2014. Proceedings, 2014. Springer. doi:10.1007/978-3-662-45174-8_15.

21 Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. *J. ACM*, 34(1):98–115, January 1987. doi:10.1145/7531.7919.

22 Fabian Frei, Ran Gelles, Ahmed Ghazy, and Alexandre Nolin. Content-oblivious leader election on rings. *CoRR*, abs/2405.03646, 2024. URL: https://arxiv.org/abs/2405.03646, doi:10.48550/arXiv.2405.03646.

23 Ran Gelles. Coding for interactive communication: A survey. *Foundations and Trends® in Theoretical Computer Science*, 13(1–2):1–157, 2017. doi:10.1561/0400000079.

24 Mohsen Ghaffari and Bernhard Haeupler. Near optimal leader election in multi-hop radio networks. In *SODA'13*, pages 748–766, New Orleans, Louisiana, USA, January 6–8, 2013, 2013. SIAM. doi:10.1137/1.9781611973105.54.

25 D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Commun. ACM*, 23(11):627–628, November 1980. doi:10.1145/359024.359029.

26 Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990. doi:10.1016/0890-5401(90)90004-2.

27 Shay Kutten, Peter Robinson, Ming Ming Tan, and Xianbin Zhu. Improved tradeoffs for leader election. In Rotem Oshman, Alexandre Nolin, Magnús M. Halldórsson, and Alkida Balliu, editors, *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC 2023, Orlando, FL, USA, June 19-23, 2023*, pages 355–365. ACM, 2023. doi:10.1145/3583668.3594576.

28 Gérard Le Lann. Distributed systems – towards a formal approach. In Bruce Gilchrist, editor, *Information Processing, Proceedings of the 7th IFIP Congress 1977*, pages 155–160, Toronto, Canada, August 8–12, 1977, 1977. North-Holland.

29 Gary L. Peterson. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Trans. Program. Lang. Syst.*, 4(4):758–762, October 1982. doi:10.1145/69622.357194.

30 Nicola Santoro and Peter Widmayer. Distributed function evaluation in the presence of transmission faults. In Tetsuo Asano, Toshihide Ibaraki, Hiroshi Imai, and Takao Nishizeki, editors, *Algorithms, International Symposium SIGAL '90*, volume 450 of *Lecture Notes in*

Computer Science, pages 358–367, Tokyo, Japan, August 16–18, 1990, 1990. Springer. doi:10.1007/3-540-52921-7_85.

31 Violet Syrotiuk and Jan Pachl. A distributed ring orientation algorithm. In *International Workshop on Distributed Algorithms (WDAG'87)*, pages 332–336. Springer Berlin Heidelberg, 1988. doi:10.1007/BFb0019813.

Sorting in One and Two Rounds Using t-Comparators

Ran Gelles
Bar-Ilan University, Ramat Gan, Israel

Zvi Lotker
Bar-Ilan University, Ramat Gan, Israel

Frederik Mallmann-Trenn
King's College London, UK

Abstract

We examine sorting algorithms for n elements whose basic operation is comparing t elements simultaneously (a t-comparator). We focus on algorithms that use only a single round or two rounds – comparisons performed in the second round depend on the outcomes of the first round comparators. Algorithms with a small number of rounds are well-suited to distributed settings in which communication rounds are costly.

We design deterministic and randomized algorithms. In the deterministic case, we show an interesting relation to design theory (namely, to 2-Steiner systems), which yields a single-round optimal algorithm for $n = t^{2^k}$ with any $k \geq 1$ and a variety of possible values of t. For some values of t, however, no algorithm can reach the optimal (information-theoretic) bound on the number of comparators. For this case (and any other n and t), we show an algorithm that uses at most three times as many comparators as the theoretical bound.

We also design a randomized Las-Vegas two-round sorting algorithm for any n and t. Our algorithm uses an asymptotically optimal number of $O(\max(\frac{n^{3/2}}{t^2}, \frac{n}{t}))$ comparators, with high probability, i.e., with probability at least $1 - 1/n$. The analysis of this algorithm involves the gradual unveiling of randomness, using a novel technique which we coin the *binary tree of deferred randomness*.

2012 ACM Subject Classification Theory of computation → Sorting and searching; Mathematics of computing → Probabilistic algorithms; Theory of computation → Distributed algorithms

Keywords and phrases Sorting, Steiner-System, Round Complexity, Deferred Randomness

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.27

Related Version *The full version of this work is available in*: `arXiv:2405.12678` [18]

Funding *Ran Gelles*: Research supported in part by the United States-Israel Binational Science Foundation (BSF) through Grant No. 2020277.
Frederik Mallmann-Trenn: Was funded by the EPSRC grant EP/W005573/1.

Acknowledgements R. Gelles would like to thank Paderborn University and CISPA – Helmholtz Center for Information Security for hosting him while part of this research was done. The authors would also like to thank the anonymous reviewers for multiple helpful comments.

1 Introduction

Sorting has been a fundamental task for computers (and earlier electronic devices) since the inception of computer history [24, 13]. Many sorting algorithms are *comparison-based*, meaning that there exists some device that compares pairs of elements and decides which of them is the larger. By comparing multiple pairs, one can obtain a full order of all elements. It is well known that if pairs are being compared, $\Theta(n \log n)$ comparisons are needed in order to fully sort any possible set of n elements. Such sorting, however, assumes one can apply

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 27; pp. 27:1–27:20
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

comparisons in an adaptive manner, i.e., one can determine which pairs of elements to compare next based on results of previous comparisons. It is not too difficult to see that without this adaptive selection of elements, $\Omega(n^2)$ comparisons are needed (see also Lemma 4 below).

In contrast to general-purpose CPUs, which allow fast comparison of two elements, specialized hardware that can be found in system-on-a-chip systems and GPUs, allows comparing larger sets of elements. Motivated by the above, in this work we explore sorting algorithms that use t-comparators. These blocks allow t elements to be compared simultaneously to determine their total order, rather than comparing them in pairs. Our initial focus is on deterministic, non-adaptive sorting algorithms where all comparisons are pre-determined and independent of prior outcomes. Additionally, we consider randomized algorithms with a limited degree of adaptiveness. In particular, we design sorting algorithms with two rounds, where the second round can use the comparison outcomes from the first round. In both cases, our goal is to minimize the number of t-comparators used.

To further motivate the case of sorting with t-comparators (t-sorting) in a single round, consider the following scenario, which is very common in the Computer Science community. A conference program committee (PC) is set to decide on the ranking of the n submitted papers. Let us assume that there is an "absolute truth", namely, that there exists a total ordering of the papers, and that each PC member outputs the "true" ordering of any number of papers assigned to them.[1] To balance out the load, the papers are split so that each PC member receives t papers. Note that the same paper can be sent to multiple PC members. Each PC member, individually, returns to the chair the total order of the set of papers assigned to them. The chair collects all these outputs and composes a total ordering of the n papers, that is consistent with all the partial sets. Assume we wish the chair's output to be the "true" ordering of the papers, how many PC members are needed, as a function of n and t? Note that the chair assigns the papers once, without having any information about papers, that is, this is a non-adaptive t-sorting with a single round.

1.1 Deterministic Sorting

Consider deterministic t-sorting algorithms with a single round. Similar to the case of $t = 2$, that requires comparing all $\binom{n}{2}$ possible pairs, it can easily be shown that for any t, at least $\gamma_{n,t} = \binom{n}{2}/\binom{t}{2}$ many t-comparators are needed in order to fully sort n elements. This stems from the fact that in order to learn the total ordering of n elements, we need to learn the relative-order of all $\binom{n}{2}$ pairs, while each t-comparator gives us information about at most $\binom{t}{2}$ different pairs of elements (Corollary 5).

Our first question is whether this bound is achievable, that is, whether there exists a single round t-sorting algorithm that utilizes exactly $\gamma_{n,t}$ comparators. We first show a way to perform t-sorting with at most $3\gamma_{n,t}$ comparators (Lemma 6). The idea is rather simple: we divide the elements into disjoint subsets, where each subset contains $t/2$ different elements. Then, we go through all possible pairs of subsets, and for each such pair we compare the t elements of their union using a separate t-comparator. This guarantees that any two elements are compared by at least one comparator, so a total-ordering of the n elements can be deduced from the results of the $\binom{\lceil n/(t/2)\rceil}{2} < 3\gamma_{n,t}$ different comparisons.

Our main result is an algorithm with an optimal level of $\gamma_{n,t}$ t-comparators for the case where t is a power of a prime and $n = t^{2^k}$, for any positive integer $k \in \mathbb{N}$. Namely,

[1] We realize that, in real life, no such absolute truth exists, and that PC members are heavily biased, etc. These extensions make a very interesting direction for followup questions. We briefly discuss future directions in Section 1.5.

▶ **Theorem 1** (main, deterministic). *Let t be a power of a prime and let $n = t^{2^k}$, $k \in \mathbb{N}$. Then, there exists a deterministic single-round, t-sorting algorithm that utilizes exactly $\binom{n}{2}/\binom{t}{2}$ comparators.*

In order to obtain the above optimal sorting, we show a connection between sorting and combinatorial design theory. Consider the case where t is a prime power and $k = 1$, that is, $n = t^2$, a setting that attained a lot of interest in the past, especially by hardware-implementation oriented designs [40, 35, 36]. We essentially show that sorting with $\gamma_{n,t}$ comparators is equivalent to an *Affine Plane* of order t. An affine plane (see e.g., [22, 31]) is a design structure composed of elements ("points") and subset of elements ("lines") that guarantees the following properties: (P1) every two points belong to a *unique* line, (P2) every line contains at least two points, and (P3) not all points are co-linear. Further, it satisfies the Euclidean Property (A1): for every line L and any point p outside L, there exists a unique line that contains p and is parallel to L. It is known that all lines in an affine plane contain exactly the same number of points; call this number *the order of the plane*. It is also known that an affine plane of order t contains $t^2 + t$ lines.

If we think about points as the elements we wish to sort and about lines as subsets of t points which we compare via a single comparator, finding an affine plane of order t provides the property that any two elements are being compared *exactly* a single time, i.e., by a single comparator, leading to the optimal bound of $\gamma_{n,t}$ comparators.

An affine plane of order t is easy to construct for any t that is a power of a prime. Let $\mathbb{F}$ be a finite field with t elements, and consider pairs of elements (x, y), i.e., the plane $\mathbb{F}^2$. In this plane, any two points (x_1, y_1) and (x_2, y_2), define a unique line that passes through them, namely $y = \frac{y_1-y_2}{x_1-x_2}x + \frac{y_2x_1-y_1x_2}{x_1-x_2}$ if $x_1 \neq x_2$ and the line $\{(x_1, y) \mid y \in \mathbb{F}\}$, otherwise. It is easy to verify this structure satisfies all the properties of an affine plane (see [31, Section 3.2]).

Affine planes are a special case of a more general combinatorial structure known as Steiner systems (Definition 10). Indeed, if we change assumption (A1) so that there exists no parallel lines at all (also known as the Elliptic Property), but still require that any two points define a unique line, we would still get a sorting algorithm in which any two elements are being compared against each other exactly once. In this case, the resulting structure is again a special case of a Steiner system known as a *Projective Plane*. Known constructions of projective planes imply that for any $t - 1$ being a power of a prime, one can sort $t^2 - t + 1$ elements using exactly $t^2 - t + 1$ many t-comparators, where every pair of elements is being compared exactly once. These two constructions are summarized as Theorem 12.

We lift the above result from optimally sorting t^2 elements to optimally sorting t^{2^k}, by developing a composition theorem (Lemma 15) that recursively performs sorting of t^{2^k} elements by utilizing an optimal number of $t^{2^{k-1}}$-comparators, for any $k > 1$.

1.2 Randomized Sorting

Similar to the deterministic case, if one does not bound the number of adaptive rounds a randomized sorting algorithm is allowed to make, optimal sorting can easily be achieved. For instance, Beigel and Gill [8] showed a generalized t-*quicksort* algorithm that sorts n elements by utilizing at most $4\frac{n \log n}{t \log t}$ many t-comparators, which is optimal, maybe up to the constant (see Theorem 16). However, this algorithm requires $O(\log_t n)$ adaptive rounds. Indeed, recall that quicksort works in rounds, where at each round the algorithm selects (one or more) pivot elements. These elements are used to "bucket" the rest of the elements into disjoint subsets, meaning that all elements greater than one pivot and less than the next pivot belong to the same bucket. Then, each such bucket is recursively sorted by the same method. Since each round depends on the pivots and buckets of the previous rounds, $O(\log_t n)$ recursive rounds are needed [8].

Our second question in this work is how to obtain optimal randomized t-sorting algorithms with restricted number of rounds. Since we already analyzed the case of a single round and reached optimal results, in the second part of this work we address the case of *two* rounds. Our goal is to minimize the number of t-comparators used to sort n elements in a Las-Vegas algorithm, where the output is correct with probability 1 but the *number of comparators* used is a random variable that varies between different instances.

Our main result for this part is as follows.

▶ **Theorem 2** (main, randomized). *Let $t < n$ be given. There exists a (Las-Vegas) randomized sorting algorithm for n elements with two rounds, that utilizes $O\left(\max\left(\frac{n^{3/2}}{t^2}, \frac{n}{t}\right)\right)$ many t-comparators, with probability at least $1 - 1/n$.*

We note that for the case where $n = t^2$, our algorithm uses $O(t)$ comparators which is asymptotically optimal since $\frac{n \log n}{t \log t} = \Theta(t)$. We further note that a result by Alon and Azar [3] implies that the expected number of comparators used in our algorithm when $t \leq \sqrt{n}$, is also tight.

The high-level idea of the two-round algorithm is to perform a single round of "quicksort" and then to optimally (deterministically) sort each resulting bucket, rather than recursively sorting it. In more details, let m be some fixed parameter. Our algorithm starts by sampling m elements that will serve as pivots. We bucket all the elements by dividing the rest $n - m$ elements into subsets of size m elements each, and comparing each such subset, along with the m pivots by utilizing at most $3\gamma_{2m,t}$ many t-comparators (per subset). This step tells us, for each one of the $n - m$ elements, between which two pivots it resides.

A pseudo code of our 2-round randomized algorithm is given below as Algorithm 1 for the case $t \leq m$. The case $t > m$ is very similar and is covered in Section 4.

Algorithm 1 A randomized 2-round sorting for any n, t with $t \leq m$.

```
Round 1:
Let $P$ be a set of $m$ elements from $A$, each sampled uniformly and independently from $A$.
Partition $A \setminus P$ into subsets $A_1, \ldots, A_k$ of size at most $m$ each.    ▷ $k = \lceil (n-m)/m \rceil$
for all $i \in [k]$ do
    Sort $P \cup A_i$ using the optimal 1-round deterministic algorithm.
end for

Round 2:
Let $P = (p_1, \ldots, p_m)$ be the ordered elements in $P$. For $1 \leq i \leq m-1$, set $S_i$ to contain all the elements which are greater than $p_i$ but lower than $p_{i+1}$. Set $S_0$ to be all the elements lower than $p_1$ and $S_m$ be all the elements greater than $p_m$.
for all $0 \leq i \leq m$ do
    Sort $S_i$ using the optimal 1-round deterministic algorithm.
end for
```

In expectation, each bucket is of size $\approx n/m$ and sorting a bucket of this size takes $3\gamma_{n/m,t}$ many t-comparators. If all buckets had size exactly n/m, this would lead immediately to the desired result of $3\frac{n}{m}\gamma_{2m,t} + 3(m+1)\gamma_{n/m,t} = O(\frac{nm}{t^2} + \frac{n^2}{mt^2})$. This quantity is minimal when

$m \approx \sqrt{n}$ (ignoring constants), leading to the claimed $O(\frac{n^{3/2}}{t^2})$ bound.[2] Unfortunately, buckets' sizes vary, and some of them might be much larger, say, of size $(n/m)\log(n/m)$. However, our analysis shows that this event is very rare and the additional number of comparators needed to handle these cases is rather small. More specifically, in our analysis, we formulate a balls-into-bins process to distribute elements into buckets, and bound the number of such bad events using the balls-into-bins process. Let us now expend on the techniques used in this analysis.

1.2.1 Techniques: The binary tree of deferred randomness

Let us start by describing the balls-into-bins process we use. Consider the n elements, and rename them $a_1, \ldots, a_n$ so that they are sorted. Starting with a_1, we group together sequences of cn/m consecutive elements, for some sufficiently large constant c. We call each such group *a bin*; namely, the first bin is $b_1 = \{a_1, \ldots, a_{cn/m}\}$ the second bin is $b_2 = \{a_{cn/m+1}, \ldots, a_{2cn/m}\}$ and so on, resulting in a total of m/c bins overall. The balls will be the m elements we pick as pivots. That is, let $P = \{p_1, p_2, \ldots, p_m\}$ be the elements selected as pivots. Since each pivot is sampled uniformly at random, the selection of some p_i is equivalent to throwing a ball to bin b_j where $p_i \in b_j$.[3]

If each bin has a ball, than each "bucket" has at most $2cn/m$ elements, and the cost, measured in the number of comparators needed to sort that bucket, is as desired. However, the absence of a ball in a bin implies larger buckets. That is, the size of the bucket, and hence the cost of sorting it, is determined by the stretch of bins without balls (up to two additional bins, one from each side). In other words, in order to bound the cost of the second round, we throw $|P| = m$ balls uniformly at random into m/c bins and count the length of consecutive *empty* bins. Recall that $m = \sqrt{n}$; we will substitute this value to avoid cumbersome equations in the following.

A straightforward balls-into-bins analysis shows that there are c pivots per bin in expectation and that the probability of not having a pivot in c' consecutive bins scales as $e^{-\Omega(c')}$. Ideally we would like to use the above probability and obtain a polynomially-small failure probability by considering all the bins at the same time. Unfortunately, this approach breaks due to the correlation between empty bins. Indeed, the fact that some bins are empty indicates that the balls went somewhere else, altering the probability of having empty bins elsewhere. The bins' loads are negatively correlated. This means that concentration bounds could potentially be used for negatively correlated variables. However, there are many obstacles to this approach. First, note that while the loads of the bins are negatively correlated, we actually need to bound different variables, namely, the lengths of consecutive sequences of empty bin. Second, defining these variables and analyzing their probability function, as well as proving that they are negatively correlated, seems to be a difficult task. Finally, note that even the number of these random variables, is itself a random variable.

Instead, we introduce the concept of a *binary tree of deferred randomness* that enables a more straightforward analysis of the concentration of empty bins, circumventing difficulties arising from their dependencies.

[2] The term $O(n/t)$ in Theorem 2 stems from the other case, where $t > m$, i.e., $t > \sqrt{n}$.

[3] We note that this balls-into-bin process differs slightly from our pivot selection process in the sense that it samples pivots with replacement, while the original process samples without replacement. However, one could modify the original process by allowing the same element to be sampled multiple times, and later ignore these extra copies. It is immediate that sampling without replacement can only create smaller bins and thus improve the overall complexity.

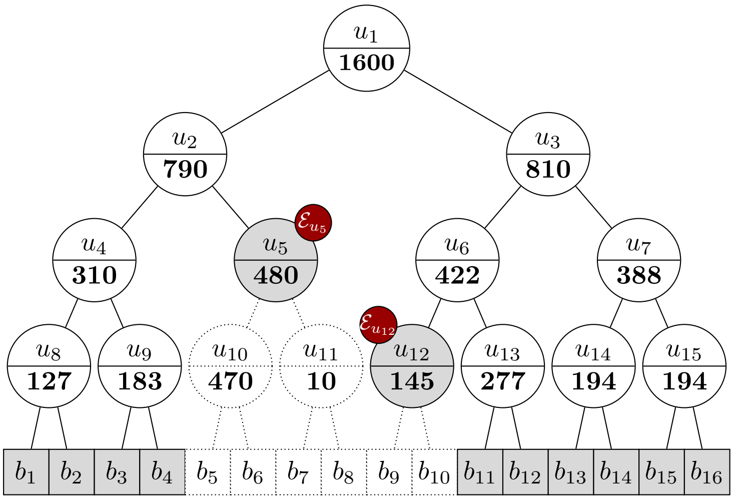


Figure 1 The figure shows the distribution of pivots (balls) on the *tree of deferred randomness*, marked as the numbers in each node. Here we have $\sqrt{n} = 1600$ pivots and 16 bins ($c = 100$). In the first two levels, the distribution is about even. The node u_{11} receives too few balls and so the event $\mathcal{E}_{u_5}$ holds. Similarly, b_9 gets too few balls (bin's balls are not shown in the figure), causing $\mathcal{E}_{u_{12}}$ to happen. The nodes in gray portray the set END described in detail in the full version.

We think of the assignment of a pivot (a ball) to a bin as the bit-string describing the bin where the pivot ends, that we reveal bit-by-bit. We define a binary tree, where each one of the $\sqrt{n}/c$ bins is a leaf. Thus, the tree has a depth of $\log(\sqrt{n}/c)$ (assuming $\sqrt{n}/c$ is a power of two). We define the following iterative process of assigning balls to the leaves of the tree: Initially we have $\sqrt{n}$ balls at the root. At every step, at every node u, we randomly assign each ball to one of u's children. This is equivalent to revealing the next bit in the string representing the bin to which the pivot belongs to. The advantage of this approach lies in the careful revelation of the randomness. At every level, we can derive concentration bounds without affecting the following levels – the only thing that matters at a given node is how many balls arrive at it.

Consider the binary tree of deferred randomness after all balls are assigned and follow an arbitrary path from the root to a leaf v. There are two cases. In the ideal case, at every node along the path to v, the number of balls going left and right is close to the expected value, namely, close to half. If this happens, then enough balls propagate along this path and with high probability at least one of them will reach the leaf v. This is the good scenario, since if this holds for many bins, the cost of sorting their elements will be very close to the expected cost.

The second case is when the concentration fails at some node u on the path, and the assignments of the balls is not close to half. If this happens first at node u, we say that the bad event $\mathcal{E}_u$ occurred, stop the process there (i.e., ignore other nodes in u's subtree), and charge a cost as if only a single ball reaches the bins under the node u. In other words, if there are ℓ balls at node u, we assume that all the ℓ pivot selections ended up picking the same element. By doing so, we overestimate the size of the resulted bucket to contain all the elements in all the bins below u. Specifically, we charge this event with the cost of sorting $O(\ell \cdot c\sqrt{n}) > |\text{bins}(u)| \cdot c\sqrt{n}$ elements; here we use the fact that, as long as the bad event $\mathcal{E}_u$ does not happen, the number of balls reaching u always exceeds the number of bins in the subtree of u, $|\text{bins}(u)|$.

Figure 1 illustrates the infiltration of balls through the tree: a node u at level i is associated to the $2^{\log(c\sqrt{n})-i}$ bins below it. The number inside a node denotes how many balls are assigned to that node. When u assigns the balls to its children, each ball picks one of the children uniformly at random, so each of the children is assigned half of u's balls, in expectation. The process continues until we reach the leaves at level $\log(c\sqrt{n})$. In the rare event that balls are distributed in a very skewed manner, the bad event $\mathcal{E}_u$ happens. For instance, while u_5 has 480 balls, they split very unevenly among its children, causing the bad event $\mathcal{E}_{u_5}$. The process stops there, i.e., we do not care how the balls continue in the subtree of u_5 and in particular, $\mathcal{E}_u$ never happens in any of u's descendants. Since $\mathcal{E}_{u_5}$ happens and the process stops there, the analysis charges an *amortized* cost which is proportional to sorting a bucket of size of 4 bins (due to the 4 bins b_5, b_6, b_7, b_8 – for all we know, all the balls could end up in b_8, creating a single bucket that consists all the respective elements). In fact, we upper bound this cost by the number of balls that arrive to u_5, whose expectation in this example is $4c \gg 4$. The situation might get even worse, since $\mathcal{E}_{12}$ occurs as well. This effectively means that a single bucket might consist of all the element in bins b_4–b_{10}. The dependency between neighboring nodes with bad events complicates the cost analysis. However, by summing up the costs of all these events, we can derive the amortized cost per such bad event and simplify the analysis by considering a single event at a time.

Luckily, the higher up in the tree a node is, the more balls the node holds and the less likely the concentration bound will fail. The lower in the tree the node is, the lower the cost is. In particular, once we approach the lower levels of the tree, the bad event $\mathcal{E}_u$ occurs with constant probability. This does not pose any trouble, because the cost in this case is only a constant factor larger than the expected cost of the case where each bin has at least one ball in it. Overall, we show that for every level of the tree, the cost imposed in our process is very close to its expectation, with high probability (at least $1 - 1/n^2$). Taking a union bound over all the (at most n) levels of the our tree of deferred randomness yields the desired claim. We give the full details in Section 4.3.

1.3 Related Work

A fundamental task like sorting naturally attracted a lot of attention in numerous variants and settings. To put our result in the right context, in this section we mention just a few of these variants and we mainly focus on *comparison-based* sorting algorithms. We refer the reader to surveys [27, 16, 38] and books [24, 13, 2] for a more complete treatment on the background of (general) sorting.

The task of sorting in small number of rounds was initiated by the work of Häggkvist and Hell [20], who considered the case of sorting n elements in a single round by comparing pairs of elements (i.e., $t = 2$). While they do not give any explicit sorting algorithm, they bound the number of 2-comparators required for sorting in d-rounds by $\Omega(n^{1+1/d})$ from below and by $O(n^{\alpha_d \log n})$ from above, for a constant α_d that monotonously decreases towards 3/2 as d grows. Specifically, for $d = 2$, they prove that the optimal number of comparisons lies within the range $(C_1 n^{3/2}, C_2 n^{5/3} \log n)$ for some constants C_1, C_2. Alon, Azar, and Vishkin [5] improved the lower bound to $\Omega(n^{1+1/d} (\log n)^{1/d})$. Alon and Azar [3, 4] lower-bounded the *average* number of comparisons by $\Omega(dn^{1+1/d})$, for any d-round algorithm with $d \leq \log n$. They also improved the upper bound to $O(n^{1+1/d} \log n)$ for a fixed d, and to $dn^{1+O(1)/d}$ for any $d \leq \log n$. Bollobás and Rosenfeld considered a relaxed sorting task, where the relative order of εn^2 pairs might still be unknown at the end. They showed that by performing $C_\varepsilon n^{3/2}$ comparisons, one can learn the order of $\binom{n}{2} - \varepsilon n^2$ pairs, where $\varepsilon \to 0$ as $C_\varepsilon \to \infty$. In contrast to the above existential bounds for 2-comparator based algorithms, our work provides *explicit* sorting algorithms. Our algorithms are efficient, they utilize t-comparators (allowing large values of t) and are asymptotically optimal, with respect to the above bounds.

Other related tasks were also considered in the literature. Alon and Azar [4] gave bounds on the number of comparisons required for approximate sorting and for selecting the median. Braverman, Mao, and Weinberg [10] considered the task of selecting the k-rank item, in a single round (and multiple rounds), and of partitioning an unordered array into the k-top and $(n-k)$-bottom elements, in a single round. Their algorithms also work in the noisy-comparison setting, where each comparison is correct with probability $2/3$. Braverman, Mao, and Peres [9] extended the above results and gave an algorithm sorting the k-top elements in small number of rounds ($d = 1, 2$ and $d \geq 3$). They also give lower and upper bounds for this task, both in the noiseless and noisy-comparison setting.

A related approach for sorting is via *sorting networks* [7, 1, 25] and in particular, sorting networks of t-comparators, a task that was raised by Knuth [24, Question 54 in Section 5.3] and examined in [1, 30, 12, 15]. These are fixed networks of comparators with n inputs (each element is an input) and n outputs (the sorted elements). One main difference between our d-round sorting and a sorting network is that in the latter, each element appears exactly once as an input. Then, any comparator that gets this element as an input must appear in a different "round". However, in a sorting algorithm, it is possible to give the same element to multiple comparators at the same round, and then form the total order out of the outcomes of all comparators.

Distributed sorting has appeared in the literature before, but it had a different meaning than the distributed sorting we consider here. Wegner [41] and Rotem, Santoro, and Sidney [34] considered the task of moving records around in a distributed network, so that they end up in a sorted manner (i.e., records that end up at the first site have keys which are strictly smaller than the records in the second site). These works mainly focused on the number of exchanged messages. We also briefly mention parallel VLSI sorting algorithms, e.g., [40, 36, 35, 23, 29]. Here the common setting is of $n \times n$ parallel processors, usually connected as a two-dimensional grid. Each processors holds one element at any given time and can transfer the element to a neighboring processors. The goal is that the elements will end up in an ordered alignment, i.e., the minimal element at the first processors, etc. This setting is somewhat similar to our case of $n = t^2$, if we think of a row or a column of processors as a single unit that can re-order the elements in that row or column according to their rank. Another sorting variant was considered by Patt-Shamir and Teplitsky [32] (building on [26]). Here, each computer starts with n records and needs to output their rankings in the global order of all n^2 records. Also unlike our task, each computer can sort any number of records that it holds (i.e., it is not limited to being a t-comparator).

As mentioned above, randomized *quicksort* with t-comparators was given by Beigel and Gill [8]. This algorithm features an optimal number of comparators, albeit it employs a large number of rounds, $d = O(\log_t n)$. A similar quicksort idea appeared earlier by Müller [28] for $t = \Omega(\log n)$, where the t-comparator is based on a systolic approach and takes $O(t)$ time to complete a single t-tuple sorting. Atallah, Frederickson, and Kosaraju [6] extended this result to the full range of t.

Mergesort with t-comparators is given in [37], and *cubesort* with t-comparators is presented in [14].

1.4 Organization

We formally state the problem of sorting with t-comparators, setting the relevant notations in Section 2. We discuss one-round deterministic sorting in Section 3. Our optimal 2-round randomized algorithm can be found in Section 4. The detailed analysis and missing proofs are deferred to the full version of this work. In Appendix A we provide some simulations comparing our 2-round randomized algorithm with the state-of-the-art $O(\log_t n)$-round t-quicksort algorithm, showing that the latter has in fact an expected number of rounds strictly larger than 4 when $n = t^2$.

1.5 Conclusions and Future Directions

In this work we studied the fundamental task of sorting n elements with t-comparators, where the sorting algorithm is limited to a small number of interactive rounds. This setting, while interesting on its own, fits in particular to distributed and parallel settings where interactive communication is very costly while computation resources are moderately costly.

We dealt with both deterministic and randomized algorithms. In the deterministic case, we established connections between optimal sorting algorithms in one round and combinatorial design theory. While this connection allows optimal sorting for certain values of n and t, it also suggests the impossibility for other values (e.g., $t = 6$). The question of the values of n, t for which optimal sorting exists is isomorphic to the long-standing combinatorial question of deciding the values of n, t for which the Steiner system $S(2, t, n)$ exists. We hope that an algorithmic approach could shed more light on this open question, e.g. through the construction of composition theorems similar to Lemma 15, or through explicit constructions for special cases.

Another interesting question is how the optimal number of t-comparators scales with the number of rounds. This topic was thoroughly examined in the literature for $t = 2$, and we extend the discussion to larger values of t. In the same vein, in the randomized setting, we design algorithms that use only two rounds but utilize the same asymptotic number of comparators as the optimal $O(\log_t n)$-round t-quicksort algorithm.

We believe our findings might be useful in other distributed settings. For instance, in the Massively Parallel Computation model (MPC), where each worker machine performs the actions of one t-comparator, and all machines act in parallel. While our algorithm for $d = 1$ rounds requires a large number of machines (i.e., more than n/t), it might make sense to consider a larger amount of rounds and how it tradeoffs the number of machines in use. For instance, could a sublinear number of machines be sufficient for $d = O(1)$ rounds?

2 Preliminaries

Notations. For a positive integer n, we let $[n]$ denote the set $\{1, 2, \ldots, n\}$. All logarithms are taken to base 2 unless otherwise noted. We say that an event happens *with high probability* in some parameter (usually, in the number of elements n), if the event occurs with probability at least $1 - 1/n^c$ for some positive constant $c \geq 1$.

Problem Statement. The elements are $A = \{a_1, a_2, \ldots, a_n\}$. Each element has a value $val(a_i) \in [n]$. We assume that all values are unique, so that for any $i \neq j$, $val(i) \neq val(j)$, and all values in $[n]$ are covered.

A t-comparator is a device that gets t elements $\{a_{i_1}, \ldots, a_{i_t}\}$ as an input, and outputs the respective order of their values. That is, it outputs a list $j_1, \ldots, j_t$ of indices, such that these are a permutation of $i_1, \ldots, i_t$ and it holds that $val(a_{j_1}) \leq val(a_{j_2}) \leq \cdots \leq val(a_{j_t})$. Note that it is allowed to give as an input the same element multiple times (hence the inequality in the $val()$ values).

A *round of sorting* is any assignment of elements to (possibly multiple) comparators. The output of a single round of sorting is defined to be the output of all the comparators in that round, i.e., the relative order between any t elements compared by some comparator.

▶ **Definition 3.** Sorting n elements in d rounds via t-comparators *is performing d rounds of sorting, where the assignment of round $i \leq d$ depends on the outputs of rounds $1, \ldots, i-1$. The assignment of elements to comparators is such that, for any possible assignment of values to the elements, there exists a* single *total ordering of the n elements that is consistent with all the outputs of the d rounds.*

We will usually care about the number of t-comparators required to sort n elements. Let us denote $\mathrm{OPT}(n,t,d)$ the minimal number of t-comparators required to sort n elements in d-rounds. In this paper we will focus on small values of d. In particular, in Section 3 we analyze the case of deterministic sorting in $d=1$ rounds. In Section 4 we discuss randomized sorting with $d=2$ rounds.

3 Sorting n elements in a single deterministic round

In this section we analyze sorting n elements with t-comparators in a single round. That is, we seek ways to assign elements to comparators that yield enough information to obtain a total-ordering of the elements. Since we restrict ourselves to a single round, we cannot adaptively select elements to compare based on previous result. Instead, all the assignments must be predetermined.

We begin with a few straightforward observations and facts. The following lemma is probably a well known folklore: if we are allowed to compare only pairs of elements ($t=2$) and the comparisons are non-adaptive ($d=1$), then *all* pairs of elements must be compared in order to obtain the total-ordering of the n element.

▶ **Lemma 4.** *For $t=2$, sorting n elements with 2-comparators in $d=1$ rounds requires learning the relative order of each of the $\binom{n}{2} = \Theta(n^2)$ pairs of elements. Thus, $OPT(n,t=2,d=1) = \binom{n}{2}$.*

Proof. Otherwise, there are two elements a_i, a_j that are not compared against each other. Let the two minimal elements (in the ranking) be a_i, a_j, respectively. Switching their relative order (i.e., letting the minimal elements be a_j, a_i, respectively) will not change the outputs of any of the comparators. Hence, there are two total ordering consistent with all the outputs, contradicting the fact that this is a sorting of n elements, Definition 3. ◀

▶ **Corollary 5.** $OPT(n,t,1) \geq \binom{n}{2}/\binom{t}{2}$.

Proof. The proof of Lemma 4 extends to larger comparators. If two elements are not being compared by some comparator, let them be of minimal value and exchange their relative order to end up with two consistent total ordering. Thus, $\mathrm{OPT}(n,t,1)$ must provide enough comparators to compare all pairs.

Each t-comparator gives the ranking of t elements among themselves. That is, it allows us to learn the (pair-wise) order between at most $\binom{t}{2}$ pairs of elements. The statement immediately follows. ◀

Note that

$$\frac{\binom{n}{2}}{\binom{t}{2}} = \frac{n(n-1)}{t(t-1)} \geq \frac{n^2}{t^2} - \frac{n}{t^2}. \tag{1}$$

We can show that sorting with at most twice the amount of optimal comparators of Eq. (1) can be achieved for certain values of n, t; sorting with at most three times the optimal is always possible.

▶ **Lemma 6.** *When $(t/2) \mid n$, $OPT(n,t,1) < 2\binom{n}{2}/\binom{t}{2}$. Otherwise, $OPT(n,t,1) < 3\binom{n}{2}/\binom{t}{2}$.*

Proof. Assume $(t/2) \mid n$. Split the n elements into $2n/t$ subsets of size $t/2$ each, $S_1, \ldots, S_{2n/t}$. Now, for any $i, j \in [2n/t]$ compare the elements in $S_i \cup S_j$ using a t-comparator. It is immediate that any two elements will be compared in this process. The total number of comparators used is

$$\binom{2n/t}{2} = \frac{1}{2} \cdot \frac{2n}{t}\left(\frac{2n}{t} - 1\right) = \frac{2n^2}{t^2} - \frac{n}{t}.$$

The above is clearly larger than twice Eq. (1), by noting that $n/t \geq 2n/t^2$ holds for $t \geq 2$.

However, when $t/2$ does not divide n, we need one additional subset $S_{2n/t+1}$ for the leftovers. This results with a total of $2n^2/t^2 + n/t$ comparators. When $t < \frac{-1+\sqrt{1+8n}}{2}$, this is still within a factor 2 of $\binom{n}{2}/\binom{t}{2}$. Otherwise, it is easy to see that we are within a factor 3 of the lower bound. Let us bound the ratio

$$\frac{\frac{2n^2}{t^2} + \frac{n}{t}}{\frac{n(n-1)}{t(t-1)}} = \frac{t-1}{t} \cdot \frac{2n+t}{n-1}.$$

The right hand side monotonically increases in t, and obtains its maximal value at $t = n-1$. This yields

$$\frac{n-2}{n-1} \cdot \frac{3n-1}{n-1}.$$

This function monotonically increases in n (as can easily be seen from its derivation) and has a limit of 3 as $n \to \infty$. ◀

3.1 The case of a large t

Let us now give optimal sorting assignments with $d = 1$ for the case of a large comparator, $t = \Omega(n)$. To demonstrate the basic idea, assume $t = n-1$. We argue that three comparators suffice in this case, which makes the bound in Lemma 6 tight for $n \geq 9$. First, we compare $\{a_1, \ldots, a_{n-1}\}$ which gives a total-ordering for all elements but the last element, a_n, so we need to compare a_n with all the other elements. This can be done with by employing two additional comparators, e.g., comparing $\{a_n, a_2 \ldots, a_{n-1}\}$ and $\{a_n, a_1, \ldots, a_1\}$. Note that the second comparator is substantially under-utilized. This means that we could still perform sorting with only three comparators even for smaller values of t.

▶ **Lemma 7.** *For any $t \geq \frac{2}{3}n$, sorting n elements in a single round can be done with three comparators.*

Proof. The inputs to the three comparators are

$$\{a_1, \ldots, a_t\}, \quad \{a_n, a_{n-1}, \ldots, a_{t+1}\ , a_1, a_2, \ldots, a_{\lceil t/2 \rceil}\}, \text{ and}$$
$$\{a_n, a_{n-1}, \ldots, a_{t+1}\ , a_{\lceil t/2 \rceil + 1}, \ldots a_t\}.$$

Note that any two elements a_i, a_j are being compared by some comparator, yielding all the information we need to obtain a single consistent total order of the elements.

Since $t \geq \frac{2}{3}n$, the second and third comparators get each $(n - (t+1) + 1) + \lceil \frac{t}{2} \rceil \leq \lfloor \frac{3}{2}t \rfloor - t + \lceil \frac{t}{2} \rceil = t$ elements as input. Note that the ceiling/flooring matters only when t is odd. In this case $\frac{3}{2}t$ is fractional and since n must be an integer, we have $n \leq \lfloor \frac{3}{2}t \rfloor$. ◀

The above three comparators construction is tight, as it is impossible to sort n elements with only two comparators. The proof resembles the approach taken by Lemma 4 for the case of $t = 2$.

▶ **Lemma 8.** *For any $t < n$, sorting in one round cannot be achieved with two comparators.*

Proof. By a pigeonhole principle, there must exist (at least) two elements a_i, a_j that are not compared against each other. We make it so $\forall k \in [n] \setminus \{i, j\}$, $val(a_i) < val(a_k)$ and $val(a_j) < val(a_k)$. Then, it is impossible to determine which one of a_i, a_j is the minimal element. Specifically, setting $val(a_i) < val(a_j)$ gives the same comparator outputs as the case where $val(a_j) < val(a_i)$. This follows since they both are lower than any other element and no comparator has both of them as input. Then, there exists two total ordering consistent with the output of the comparators: one with $val(a_i) < val(a_j)$ and the other with $val(a_j) < val(a_i)$, contradicting Definition 3. ◀

3.2 Minimal sorting for a variety of parameters via design theory

Recall the proof of Corollary 5. It implies that every two elements must be compared against each other. This leads us to defining *minimal* sorting as follows.

▶ **Definition 9.** *Sorting is said to be* minimal *if equality holds in the equation in Corollary 5.*

That is, minimality is obtained when every two elements are compared against each other *exactly* once, and all the t-comparators are fully utilized. Then on the one hand there is no redundancy, and on the other hand all computational resources are fully used. Note that *optimality* means the minimal number of comparators needed to get all pairs compared against each other exactly once, but without requiring that all comparators are fully utilized.

While minimality implies optimality, the other direction does not hold. As demonstrated above for $2n/3 \leq t < n$, optimality is obtained with 3 comparators. However, minimality is not obtainable in this case. For instance, when $n = 10$, and $t = 7, 8, 9$ we have $\binom{n}{2}/\binom{t}{2} \in [1\frac{1}{4}, 2\frac{1}{7}]$, but, as we proved, exactly 3 comparators are necessary in all these cases, i.e., some comparator must be under-utilized regardless of the sorting algorithm.

▶ **Definition 10** (A Steiner System). *A* Steiner System *with parameters $0 < c < t < n$, denoted $S(c, t, n)$, is a set $\mathbb{P}$ of n elements (we will call* points*) and a set $\mathbb{L}$ of objects (we will call* lines*), where each line is a subset of t points and it holds that any subset of c points is contained in exactly a single line.*

Corollary 5 and the discussion above imply the following.

▶ **Theorem 11.** *The Steiner system $S(2, t, n)$ is equivalent to a minimal sorting of n elements via t-comparators.*

Proof. Immediate from definitions. Every point is an element to sort, every line is a single comparator. Since any two points are contained exactly in a single line and since every line contains exactly t points, we obtain minimality. ◀

The above equivalence allows us to use known results about $S(2, t, n)$ to deduce cases for which minimal sorting is possible. The following is an immediate corollary of the known state-of-the-art about Steiner systems with $c = 2$, see e.g., [22, 19, 33].

▶ **Theorem 12.**

1. *Let t be a power of a prime. Minimal sorting of $n = t^2$ elements is possible by employing $t^2 + t$ many t-comparators.*
2. *Let $t - 1$ be a power of a prime. Minimal sorting of $n = t^2 - t + 1$ elements is possible by employing $t^2 - t + 1$ many t-comparators.*

Proof. (1) Follows from the fact that every field of size t implies a Steiner system $S(2, t, t^2)$ (an Affine Plane), see [31, Section 3.2]. (2) Follows from the fact that every field of size $t - 1$ implies a Steiner system $S(2, t, t^2 - t + 1)$ (a Projective Plane), see [31, Section 4.5]. We note that both constructions are explicit. ◀

The equivalence stated in Theorem 11 also yields some impossibilities on minimal sorting. It is well known that the Steiner system $S(2, 6, 36)$ does not exist. This problem, stated originally as a question about Latin Squares and known as the 36 officers problem, dates back to Euler [17] and was proven impossible by Terry [39]. Bruck and Ryster [11] extended this result and proved that Steiner systems of many other orders are also impossible.

▶ **Corollary 13** ([39, 11])**.** *Minimal sorting of $n = t^2$ elements (i.e., with exactly $t + t^2$ many t-comparators) is impossible for infinitely many values of t.*

Despite decades of research, a full characterization of values of t that admit a $S(2, t, t^2)$ system does not exist. In 1975, Willson [42] showed that for any t, a Steiner $S(2, t, n)$ system exists if and only if $t \mid n$ and $t(t - 1) \mid n(n - 1)$, *except for finitely many values of n*. This implies the following corollary

▶ **Corollary 14.** *For any t and large enough integer c, minimal sorting of $n = t^c$ elements is possible with $OPT(n, t, 1) = \binom{n}{2}/\binom{t}{2}$ many t-comparators.*

Indeed, for any $c \geq 1$ we have that $t \mid t^c$ and $(t - 1) \mid (t^c - 1)$ since $t^c - 1 = (t - 1)(t^{c-1} + t^{c-2} + \cdots + 1)$. Our composition theorem, which is given in the next section (Lemma 15), gives explicit construction for some values of n, t. Finding explicit constructions for other values remains open.

3.3 A Composition Theorem

The above Theorem 12 applies only to the cases where $n = t^2$ or $n = t^2 - t + 1$ (for certain values of t). An interesting question is how to obtain a single-round sorting for other values of t and n, e.g., for $n = t^c$ elements, with $c \geq 3$. We partially answer this task by constructing a t^2-comparator out of an optimal number of t-comparators. Operating recursively on larger n's, this approach leads to the following theorem.

▶ **Lemma 15.** *Let t be power of a prime and let $n = t^{2^k}$ for some $k \in \mathbb{N}$. Then, minimal sorting of n elements with t-comparators is possible and employs $OPT(n = t^{2^k}, t, 1) = \binom{n}{2}/\binom{t}{2}$ many t-comparators.*

Proof. We prove that minimal sorting is possible by induction on k. The base case, $k = 1$ is given by Theorem 12(1).

For the induction step, assume we can sort $n' = t^{2^{k-1}}$ elements using $\binom{n'}{2}/\binom{t}{2}$ many t-comparators. We show how to sort $n = t^{2^k}$ elements with exactly $\binom{n}{2}/\binom{t}{2}$ t-comparators. Since n' is a power of a prime, Theorem 12 provides us a optimal (minimal) way to sort n elements using n'-comparators. Each n'-comparator can be implemented via an optimal (minimal) number of t-comparators, by induction. The total number of t-comparator thus required to sort n elements is

$$\frac{\binom{n}{2}}{\binom{n'}{2}} \cdot \frac{\binom{n'}{2}}{\binom{t}{2}} = \frac{\binom{n}{2}}{\binom{t}{2}},$$

and this quantity is minimal by Corollary 5. ◀

As a corollary, the above composition theorem implies an explicit construction of a $S(2, t, t^{2^k})$ system for t a power of a prime and all integers $k > 0$.

4 Optimally sorting n elements in $d = 2$ randomized rounds

In Section 3, we studied optimal deterministic sorting in $d = 1$ rounds. We now wish to turn to the case of $d = 2$ rounds, trading-off one additional round for fewer comparisons. We study the *randomized* case since it allows us to reduce the number of comparisons considerably. Since for $d = 1$ we have already obtained an optimal deterministic solution, it makes sense to discuss randomized algorithms for $d > 1$. As randomized sorting with $O(\log n)$ comparators are well-known [21, 8], we wish to keep the number of rounds small, and focus on the case of $d = 2$. We design a fast randomized t-sorting algorithm, which is asymptotically optimal in the number of t-comparators used, restricted to algorithms with $d = 2$ rounds. In certain cases, for instance when $n = t^2$, the asymptotic number of t-comparators is optimal even without the round restriction. We discuss lower bounds on the number of t-comparators required for sorting in Section 4.1. In Section 4.2 we consider the special case of $d = 2$ and $n = t^2$ and in Section 4.3 we consider the more general case of $d = 2$ and arbitrary n and t. The detailed analysis is deferred to the full version. Our main result is Theorem 2, which we now recall.

▶ **Theorem 2** (main, randomized). *Let $t < n$ be given. There exists a (Las-Vegas) randomized sorting algorithm for n elements with two rounds, that utilizes $O\left(\max\left(\frac{n^{3/2}}{t^2}, \frac{n}{t}\right)\right)$ many t-comparators, with probability at least $1 - 1/n$.*

4.1 Lower bounds

Before describing our algorithms, let us recall the lower bound on the number of t-comparators, by Beigel and Gill [8].

▶ **Theorem 16** ([8]). *Sorting n elements requires utilizing at least $\frac{\log(n!)}{\log(t!)} = \frac{n \log n}{t \log t}(1 + o(1))$ many t-comparators.*

The proof stems from the fact that $\log(n!)$ bits of information are required to sort n elements, and that each comparator gives $\log(t!)$ bits of information. See Section II in [8].

The above lower bound allows any number of rounds. Alon and Azar [3] analyzed the average number of 2-comparators required to sort n elements in d rounds and proved the following.

▶ **Theorem 17** ([3]). *Sorting n elements in $d \le \log n$ rounds, requires utilizing at least $\Omega(dn^{1+1/d})$ many 2-comparators* on average.

The above theorem could be used to derive lower bounds on sorting with t-comparators. Recall that each t-comparator compares at most $\binom{t}{2}$ pairs of elements. Then, the following lower bounds on the average number of t-comparators required in any randomized sorting is immediate.

▶ **Corollary 18.** *Sorting n elements in $d \leq \log n$ rounds, requires utilizing at least $\Omega(dn^{1+1/d}/t^2)$ many t-comparators* on average.

Because any average-case lower bound is also a worst-case lower-bound, if we plug in $d = 2$ in the above corollary, we obtain that our algorithm with $O(n^{3/2}/t^2)$ many t-comparator when $t < \sqrt{n}$, is asymptotically tight.

4.2 The simple special case of $n = t^2$

In this section we present Algorithm 2, which performs t-sorting of $n = t^2$ elements in two rounds and utilizes $O(t)$ many t-comparators. Note that by Theorem 16, this is asymptotically tight, even without the restriction to $d = 2$ rounds. Although our Algorithm 3 and Algorithm 4 described in Section 4.3 are strictly more general, as they apply to any n, t, for pedagogical reasons we first introduce the simplified and very natural Algorithm 2 that assumes the special case of $n = t^2$.

Algorithm 2 A randomized 2-round sorting of $n = t^2$ elements with $O(t)$ many t-comparators.

```
Round 1:
Let $P$ be a set of $t/2$ elements from $A$, each sampled uniformly and independently from $A$.
Partition $A$ into subsets $A_1, \dots, A_k$ of size $t/2$ each.
for all $i \in [k]$ do
    Input $P \cup A_i$ into a comparator.                         ▷ $k$ comparators
end for

Round 2:
Let $P = (p_1, \dots, p_{t/2})$ be the ordered elements in $P$. For $1 \leq i \leq t/2 - 1$, set $S_i$ to
  contain all the elements which are greater than $p_i$ but lower than $p_{i+1}$. Set $S_0$ to be all
  the elements lower than $p_1$ and $S_{t/2}$ be all the elements greater than $p_{t/2}$.
for all $0 \leq i \leq t/2$ do
    Sort $S_i$ via Lemma 6.                  ▷ at most $\sum_i 3|S_i|^2/t^2$ comparators
end for
```

Recall our notations, where we wish to sort a set of $n = t^2$ elements, denoted $A = \{a_1, \dots, a_n\}$. We assume that t is even and that $(t/2) \mid n$, and set $k = n/(t/2)$. The algorithm works as follows. In the first round, we first sample $t/2$ elements uniformly from A. These will be ours “pivots”. We then take the remaining elements of A and compare them to the pivots. That is, we split the remaining elements into $n/(t/2) - 1$ disjoint subsets of size $t/2$. We input each subset to a t-comparator together with (all) the $t/2$ pivots. After this step, for each element in A we know its relative position with respect to the pivots. Since we used the same pivots in each comparator, we can see the first round as the pivots splitting A into $t/2 + 1$ disjoint “buckets” such that all the elements in one bucket are strictly smaller (or strictly larger) than all elements in any other bucket. In the second round of the algorithm, we sort each bucket separately.

The first step utilizes $n/(t/2) = 2t$ comparators, one for each subset of A. In the second part, the number of comparators in use depends on the size of the buckets we need to sort, which is a random variable determined by the pivots we sample in the first round. In expectation, each bucket is of size approximately[4] $n/(t/2+1) \approx 2\frac{n}{t}$. If we assumed that the number of elements per bin is tightly concentrated around its mean, then we could deduce that sorting a single bucket using Lemma 6 would take $O(\frac{n^2}{t^4}) = O(1)$ comparators, and summing up over all $t/2+1$ buckets results in $O(\frac{n^2}{t^3}) = O(t)$ comparators overall, in expectation.

However, we cannot make such an assumption, since, while each bucket has $\approx 2\frac{n}{t}$ elements in expectation, there might be very large buckets, with, say, $O(\frac{n}{t}\log n)$ elements. Our analysis (which we perform only to the general case, in Section 4.3 below), is somewhat more intricate and shows that the event of a large bucket is rare enough so that amortizing across all the buckets, our algorithm still takes $O(t)$ comparators with high probability.

4.3 The general case: supporting any n, t

Algorithm 2 can be executed with any n, t. The problem is that this would come at a very high cost (measured in the number of t-comparators used). The main reason for this high cost is that Algorithm 2 has a tradeoff between the costs of the different rounds: the cost of the first rounds is $O(\frac{n}{t})$ and the cost of the second is $O(\frac{n^2}{t^3})$. While these two costs equal $O(t)$ for $n = t^2$, for arbitrary n and t these costs are no longer balanced and one of the rounds would have a relatively high cost. The idea behind Algorithm 3 depicted below,[5] is to balance the costs of the phases, by carefully choosing the size of the pivot set and, as a result, the expected sizes of the buckets they yield.

Algorithm 3 A randomized 2-round sorting for any n, t with $t \le \sqrt{n}$.

```
Round 1:
Let $P$ be a set of $m = \sqrt{n}$ elements from $A$, each sampled uniformly and independently from $A$.
Partition $A \setminus P$ into subsets $A_1, \ldots, A_k$ of size at most $m$ each.    ▷ $k = \lceil (n-m)/m \rceil$
for all $i \in [k]$ do
    Sort $P \cup A_i$ via Lemma 6.
end for

Round 2:
Let $P = (p_1, \ldots, p_m)$ be the ordered elements in $P$. For $1 \le i \le m-1$, set $S_i$ to contain all the elements which are greater than $p_i$ but lower than $p_{i+1}$. Set $S_0$ to be all the elements lower than $p_1$ and $S_m$ be all the elements greater than $p_m$.
for all $0 \le i \le m$ do
    Sort $S_i$ via Lemma 6.
end for
```

Assume that the first round randomly selects m pivots, which we denote by the set P. In order to "bucket" the n elements according to the pivots we need to compare them all with all the pivots. To that end, we split the set A into subsets $A_1, \ldots, A_k$ of size m (maybe except

[4] To bound the expected size of each bucket, consider the sorted array of elements and uniformly select t pivots. Connect the beginning of the array to its end to form a cycle. Now consider all intervals between the pivots. The expected sum of the intervals, is roughly n. By linearity of expectation, we can consider disjoint "chunks" of intervals, each composed of t consecutive intervals. By symmetry, the expected lengths of all chunks are the same. Thus, each chunk must be, in expectation, about n/t elements long (ignoring constants).

[5] Algorithm 3 is identical to Algorithm 1 described in the introduction and repeated here for convenience.

for the last subset), and compare each subset with the pivots. In contrary to Algorithm 2, we can no longer input $A_i \cup P$ into a t comparator. Instead, we need to implement a $2m$-comparator out of t-comparators. We do so via Lemma 6, at the cost of $8m^2/t^2$ many t-comparators for a single simulated $2m$-comparator.

Let us now analyze the expected cost of Algorithm 3. To calculate the cost of the first round, note that we now need $n/(2m)$ many (simulated) $2m$-comparators each costing us $O(m^2/t^2)$ many t-comparators. Thus, the first round results in a total cost of $O(\frac{nm}{t^2})$. The expected cost of the second round is given as follows: since the set of pivots is sampled uniformly, the expected size of each bucket is $\approx 2n/t$ (Footnote 4). Oversimplifying again and assuming the number of elements per bin is tightly concentrated (which is not necessarily true for each bin), we get the following. By Lemma 6, each one of the $m+1$ buckets costs $O((n/m)^2/t^2)$ comparators in expectation. Overall, the expected cost in the second round is $O(\frac{n^2}{mt^2})$. Summing the costs of the two rounds, the expected cost of Algorithm 3 is $O(\frac{nm}{t^2} + \frac{n^2}{mt^2})$. Interestingly, this value is minimized when $m = \sqrt{n}$, irrespective of t. In the reminder, we simply set $m = \sqrt{n}$, and the cost becomes $O(\frac{n^{3/2}}{t^2})$.

The case of $t > \sqrt{n}$. The above analysis needs a little tweak to support the case of $t > \sqrt{n}$. In this case, the number of comparators-per-bucket given by the terms $O(m^2/t^2)$ and $O(n/m^2t^2)$ for the first and second round, respectively, is lower bounded by a single comparator, and thus should read $\max\{1, O(m^2/t^2)\}$ and $\max\{1, O(n/m^2t^2)\}$, respectively. Therefore, the choice of parameters needs to be adjusted. In the following we show a selection of parameters that optimize the case of $t > \sqrt{n}$, which yields Algorithm 4. We only give here a sketch of the (simplified) expected cost analysis, since the precise high-probability analysis see the full version.

Algorithm 4 A randomized 2-round sorting for any n, t with $t > \sqrt{n}$.

```
Round 1:
$\tilde{m} = \lceil n/t \rceil$
Let $P$ be a set of $\tilde{m}$ elements from $A$, each sampled uniformly and independently from $A$.
Partition $A \setminus P$ into subsets $A_1, \dots, A_k$ of size at most $t$ each.    ▷ $k = \lceil (n - \tilde{m})/t \rceil$
for all $i \in [k]$ do
    Sort $P \cup A_i$ via Lemma 6.
end for

Round 2:
Let $P = (p_1, \dots, p_{\tilde{m}})$ be the ordered elements in $P$. For $1 \le i \le \tilde{m} - 1$, set $S_i$ to contain
  all the elements which are greater than $p_i$ but lower than $p_{i+1}$. Set $S_0$ to be all the
  elements lower than $p_1$ and $S_{\tilde{m}}$ be all the elements greater than $p_{\tilde{m}}$.
for all $0 \le i \le \tilde{m}$ do
    Sort $S_i$ via Lemma 6.
end for
```

In Algorithm 4, We set the number of pivots to be $\tilde{m} = \lceil n/t \rceil$, and group the rest of the elements into subsets $\{A_i\}$ of size t each (instead of size $\tilde{m}$). We then continue with the sorting as before.

In the first round of the algorithm, we sort $k = O(n/t)$ sets, each of size $t + \tilde{m} = O(t)$. Thus, by Lemma 6 sorting each such bucket can be done using $c' = O(1)$ comparators resulting in $c'k = O(n/t)$ comparators in total. In the second round, each S_i has $O(n/\tilde{m}) = O(t)$ elements, in expectation. Assuming again our oversimplification that the number of elements in each bin is tightly concentrated around its mean, we get by Lemma 6 that sorting each S_i takes $O(1)$ comparators. Since there are $\tilde{m}+1$ such sets, the total number of comparators used in the second round is also bounded by $O(n/t)$.

In the full version we formally analyze the number of comparators used by these schemes (without the oversimplifying assumption) and show that it is concentrated around the stated value, i.e., we prove Theorem 2. As mentioned above, we only analyze Algorithm 3 since the analysis of Algorithm 4 is analogous. We stress again that the expected analysis presented above is oversimplified. Further, even with a simple and straightforward expected analysis, the dependencies of the events make it difficult to obtain high-probability concentration bounds, i.e., bounds that hold except with a polynomially small probability.

References

1 M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, 1983. doi:10.1145/800061.808726.

2 Selim G Akl. *Parallel sorting algorithms*, volume 12. Academic press, 1985.

3 N. Alon and Y. Azar. The average complexity of deterministic and randomized parallel comparison sorting algorithms. In *28th Annual Symposium on Foundations of Computer Science (SFCS 1987)*, pages 489–498, 1987. doi:10.1109/SFCS.1987.54.

4 Noga Alon and Yossi Azar. Sorting, approximate sorting, and searching in rounds. *SIAM Journal on Discrete Mathematics*, 1(3):269–280, 1988. doi:10.1137/0401028.

5 Noga Alon, Yossi Azar, and Uzi Vishkin. Tight complexity bounds for parallel comparison sorting. In *27th Annual Symposium on Foundations of Computer Science (SFCS 1986)*, pages 502–510, 1986. doi:10.1109/SFCS.1986.57.

6 Mikhail J. Atallah, Greg N. Frederickson, and S.Rao Kosaraju. Sorting with efficient use of special-purpose sorters. *Information Processing Letters*, 27(1):13–15, 1988. doi:10.1016/0020-0190(88)90075-0.

7 K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, 1968. doi:10.1145/1468075.1468121.

8 Richard Beigel and John Gill. Sorting n objects with a k-sorter. *IEEE Transactions on Computers*, 39(5):714–716, 1990. doi:10.1109/12.53587.

9 Mark Braverman, Jieming Mao, and Yuval Peres. Sorted top-k in rounds. In *Proceedings of the Thirty-Second Conference on Learning Theory*, volume 99 of *PMLR*, pages 342–382, 2019. URL: https://proceedings.mlr.press/v99/braverman19a.html.

10 Mark Braverman, Jieming Mao, and S. Matthew Weinberg. Parallel algorithms for select and partition with noisy comparisons. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 851–862, 2016. doi:10.1145/2897518.2897642.

11 Richard H. Bruck and Herbert J. Ryser. The nonexistence of certain finite projective planes. *Canadian Journal of Mathematics*, 1(1):88–93, 1949. doi:10.4153/CJM-1949-009-2.

12 YB Chiang. *Sorting networks using k-comparators.* PhD thesis, University of Cape Town, 2001. URL: http://hdl.handle.net/11427/4871.

13 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 4th edition, 2022.

14 Robert Cypher and Jorge L.C. Sanz. Cubesort: A parallel algorithm for sorting n data items with s-sorters. *Journal of Algorithms*, 13(2):211–234, 1992. doi:10.1016/0196-6774(92)90016-6.

15 Natalia Dobrokhotova-Maikova, Alexander Kozachinskiy, and Vladimir Podolskii. Constant-Depth Sorting Networks. In *14th Innovations in Theoretical Computer Science Conference (ITCS 2023)*, volume 251 of *LIPIcs*, pages 43:1–43:19, 2023. doi:10.4230/LIPIcs.ITCS.2023.43.

16 Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, December 1992. doi:10.1145/146370.146381.

17 Leonhard Euler. Recherches sur un nouvelle espéce de quarrés magiques. *Verhandelingen uitgegeven door het zeeuwsch Genootschap der Wetenschappen te Vlissingen*, pages 85–239, 1782.

18 Ran Gelles, Zvi Lotker, and Frederik Mallmann-Trenn. Sorting in one and two rounds using t-comparators. *CoRR*, abs/2405.12678, 2024. arXiv:2405.12678, doi:10.48550/arXiv.2405.12678.

19 Mike Grannell and Terry Griggs. An introduction to steiner systems. *Mathematical Spectrum*, 26(3):74–80, 1994.

20 Roland Häggkvist and Pavol Hell. Parallel sorting with constant time for comparisons. *SIAM Journal on Computing*, 10(3):465–472, 1981. doi:10.1137/0210034.

21 C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, January 1962. doi:10.1093/comjnl/5.1.10.

22 D. R. Hughes and F. Piper. *Design Theory.* Cambridge University Press, 1985.

23 Christos Kaklamanis and Danny Krizanc. Optimal sorting on mesh-connected processor arrays. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, pages 50–59, 1992. doi:10.1145/140901.140907.

24 Donald E. Knuth. *Art of computer programming, volume 3: Sorting and Searching.* Addison-Wesley Professional, 2nd edition, April 1998.

25 Tom Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, 1985. doi:10.1109/TC.1985.5009385.

26 Christoph Lenzen and Roger Wattenhofer. Tight bounds for parallel randomized load balancing: extended abstract. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 11–20, 2011. doi:10.1145/1993636.1993639.

27 W. A. Martin. Sorting. *ACM Comput. Surv.*, 3(4):147–174, December 1971. doi:10.1145/356593.356594.

28 Heinrich Müller. Sorting numbers using limited systolic coprocessors. *Information Processing Letters*, 24(6):351–354, 1987. doi:10.1016/0020-0190(87)90109-8.

29 S. Olarin and S.Q. Zheng. Sorting n items using a p-sorter in optimal time. In *Proceedings of SPDP '96: 8th IEEE Symposium on Parallel and Distributed Processing*, pages 264–272, 1996. doi:10.1109/SPDP.1996.570343.

30 Bruce Parker and Ian Parberry. Constructing sorting networks from k-sorters. *Information Processing Letters*, 33(3):157–162, 1989. doi:10.1016/0020-0190(89)90196-8.

31 Abraham Pascoe. Affine and projective planes. Master's thesis, Missouri State University, 2018. MSU Graduate Theses. 3233. https://bearworks.missouristate.edu/theses/3233.

32 Boaz Patt-Shamir and Marat Teplitsky. The round complexity of distributed sorting: extended abstract. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 249–256, 2011. doi:10.1145/1993806.1993851.

33 Colin Reid and Alex Rosa. Steiner systems $s(2,4,v)$-a survey. *The Electronic Journal of Combinatorics*, pages DS18–Feb, 2012.

34 Doron Rotem, Nicola Santoro, and Jeffrey B. Sidney. Distributed sorting. *IEEE Transactions on Computers*, C-34(4):372–376, 1985. doi:10.1109/TC.1985.5009389.

35 Isaac D. Scherson, Sandeep Sen, and Adi Shamir. Shear sort: a true two-dimensional sorting technique for VLSI networks. In *International Conference on Parallel Processing*, pages 903–908, 1986.

36 Claus-Peter Schnorr and Adi Shamir. An optimal sorting algorithm for mesh connected computers. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 255–263, 1986. doi:10.1145/12130.12156.

37 Feng Shi, Zhiyuan Yan, and Meghanad Wagh. An enhanced multiway sorting network based on n-sorters. In *2014 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 60–64, 2014. doi:10.1109/GlobalSIP.2014.7032078.

38 Dhirendra Pratap Singh, Ishan Joshi, and Jaytrilok Choudhary. Survey of gpu based sorting algorithms. *International Journal of Parallel Programming*, 46:1017–1034, 2018. doi:10.1007/S10766-017-0502-5.

39 G. Tarry. Le problème de 36 officiers. *Compte Rendu de l'Association Française pour l'Avancement de Science Naturel*, 1900. vol. 1 (1900), 122-123; vol. 2 (1901), 170-203.

40 C. D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. *Commun. ACM*, 20(4):263–271, April 1977. doi:10.1145/359461.359481.

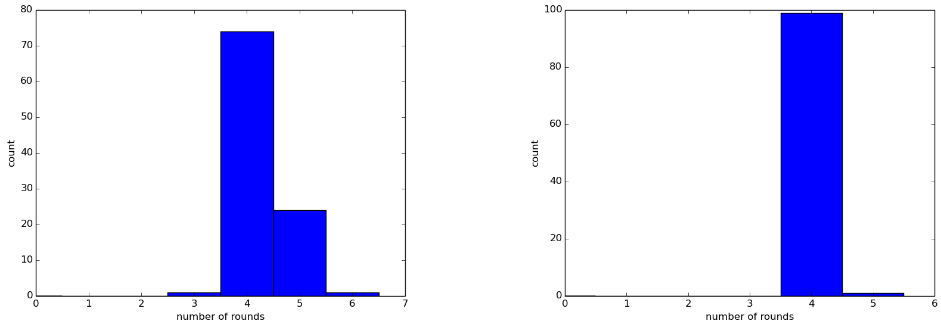
41 Lutz M. Wegner. Sorting a distributed file in a network. *Computer Networks (1976)*, 8(5):451–461, 1984. doi:10.1016/0376-5075(84)90007-2.

42 Richard M. Wilson. An existence theory for pairwise balanced designs, III: Proof of the existence conjectures. *Journal of Combinatorial Theory, Series A*, 18(1):71–79, 1975. doi:10.1016/0097-3165(75)90067-9.

APPENDIX

A Simulations: Our algorithm and the state-of-the-art algorithm

Let us compare our Algorithm 1 to the state-of-the-art quicksort algorithm with t-comparators, developed by Beigel and Gill [8]. Their algorithm works essentially as follows: randomly select $t/\log t$ pivot elements and use them to split all the elements into disjoint subsets. Now, recursively sort any subset of size exceeding t.



(a) $t = 10$, $n = 100$.

(b) $t = 100$, $n = 10000$.

Figure 2 A histogram of the number of rounds required to the completion of the algorithm in [8] for the case of $n = t^2$ with (a) $t = 10$ and (b) $t = 100$. Each histogram is based on 100 repeated independent instances. In both t values, the average number of rounds is above 4.

The analysis in [8] proves that the number of t-comparators utilized throughout this algorithm is $\frac{n \log n}{t \log t}(1 + o(1))$, which is asymptotically optimal. The same analysis suggests the algorithm takes $\log_{m/2}(n)$ rounds, where $m = t/(2\log(t)\ln(t))$. (The basis of the log in m is not defined in [8] and we take it to base e, yielding $m = t/2\ln^2 t$.) It is easy to verify that this function approaches $\frac{\log n}{\log t}$ rounds, for sufficiently large t. In particular, for $n = t^c$, the function approaches c rounds as $t \to \infty$. We would like to compare this to our algorithm, that guarantees $d = 2$ rounds, regardless of t.

To be concrete, let us consider the case of $n = t^2$. In this case, $\log_{m/2}(t^2)$ tends asymptotically to 2 when $t \to \infty$. To demonstrate the behavior of the recursive algorithm we have performed Monte-Carlo simulations that measure the number of rounds it takes to sort $n = t^2$ elements, with $t = 10$ and $t = 100$. The results are depicted in Figure 2. Our findings indicate that, for these values of t, the average number of rounds for $n = t^2$ is not 2, but rather 4.

Self-Stabilizing MIS Computation in the Beeping Model

George Giakkoupis
Inria, Univ Rennes, CNRS, IRISA, France

Volker Turau
Institute of Telematics, Hamburg University of Technology, Germany

Isabella Ziccardi
Bocconi University, BIDSA, Milan, Italy

Abstract

We consider self-stabilizing algorithms to compute a Maximal Independent Set (MIS) in the extremely weak *beeping* communication model. The model consists of an anonymous network with synchronous rounds. In each round, each vertex can optionally transmit a signal to all its neighbors (beep). After the transmission of a signal, each vertex can only differentiate between no signal received, or at least one signal received. We also consider an extension of this model where vertices can transmit signals through two distinguishable beeping channels. We assume that vertices have some knowledge about the topology of the network.

We revisit the not self-stabilizing algorithm proposed by Jeavons, Scott, and Xu (2013), which computes an MIS in the beeping model. We enhance this algorithm to be self-stabilizing, and explore three different variants, which differ in the knowledge about the topology available to the vertices and the number of beeping channels. In the first variant, every vertex knows an upper bound on the maximum degree Δ of the graph. For this case, we prove that the proposed self-stabilizing version maintains the same run-time as the original algorithm, i.e., it stabilizes after $O(\log n)$ rounds w.h.p. on any n-vertex graph. In the second variant, each vertex only knows an upper bound on its own degree. For this case, we prove that the algorithm stabilizes after $O(\log n \cdot \log\log n)$ rounds on any n-vertex graph, w.h.p. In the third variant, we consider the model with two beeping channels, where every vertex knows an upper bound of the maximum degree of the nodes in the 1-hop neighborhood. We prove that this variant stabilizes w.h.p. after $O(\log n)$ rounds.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Maximal Independent Set, Self-Stabilization, Beeping Model

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.28

Funding *George Giakkoupis*: Supported by Agence Nationale de la Recherche (ANR), under project ByBloS (ANR-20-CE25-0002).
Isabella Ziccardi: Supported by the European Research Council (ERC), under the European Union's Horizon 2020 research and innovation program (grant agreement No. 834861).

1 Introduction

The Maximal Independent Set (MIS) problem has a central role in the areas of parallel and distributed computing. In a graph $G = (V, E)$, an MIS is a subset of vertices $I \subseteq V$ where no two vertices in I are adjacent, and it is maximal with respect to inclusion. Recognized for its importance in the field of distributed computing since the early 1980s [21, 3], the computation of an MIS serves as a foundational subroutine in various algorithms in wireless networks, routing, and clustering [23]. The interest in the MIS problem has recently extended to biological networks, with observations of processes similar to the MIS elections in the development of the fly's nervous system [2].

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 28; pp. 28:1–28:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

While distributed MIS algorithms are well-explored in the standard synchronous message-passing models like LOCAL, CONGEST, and CONGESTED-CLIQUE [23, 19, 20, 12, 15, 11, 4, 14], recently the MIS selection was considered also within weaker communication frameworks [22, 1, 8]. Indeed, novel distributed communication models, inspired by scenarios in biological cellular networks, wireless sensor networks and networks with sub-microprocessor devices, were defined. The Stone Age model, introduced by Emek and Wattenhofer, provides an abstraction of a network of randomized finite state machines that communicate with their neighbors using a fixed message alphabet based on a weak communication scheme [10]. Another related model, which is the one we consider in this paper, is the full-duplex beeping model[1], where a network of anonymous processors and synchronous rounds is considered [5]. In each round, each vertex has the option to either broadcast a signal – a beep – to all its neighbors or to remain silent. Subsequently, each vertex can determine whether it received any signals or if all its neighbors remained silent. This does not allow a vertex to differentiate which vertex emitted the signal, nor the number of signals received. We notice that a variation of this model can be defined where, instead of a single type of signal, a constant number of distinct signals exist, and the vertices can distinguish between the types of signals received. The beeping model finds motivation in scenarios such as wireless sensor networks or biological systems, where organisms can only detect proteins transmitted by neighboring entities [1]. The problem of computing an MIS was already considered in the full-duplex beeping model [18, 13, 1] and in the Stone Age model [9, 8, 10].

In both biological and wireless systems, another notable trait is their capability for self-recovery. This ability is also essential in distributed and large-scale systems, which must be able to effectively manage faults. Self-stabilizing algorithms are designed to ensure that systems can recover from any state and eventually stabilize into a valid state, maintaining stability as long as faults are absent [6, 7]. Indeed, self-stabilizing algorithms are guaranteed to converge from any initial configuration. Two self-stabilizing MIS algorithms in the standard message-passing model are proposed in [16]. However, only a few self-stabilizing MIS algorithms have been proposed for the aforementioned weak communication models [1, 8, 17]. In the full-duplex beeping model, Afek et al. in [1] introduced a self-stabilizing algorithm that converges to an MIS in $O(\log^2 N \log n)$ rounds with high probability (w.h.p.), if all vertices know an upper bound N on the network's size n. They also established a polynomial lower bound for the MIS in a similar model. This model includes an adversary able to select the wake-up time slots for the vertices. Because of the presence of the adversary, the lower bound of [1] is not applicable in the setting of this paper. In the full-duplex beeping model, a constant-state algorithm was proposed in [17], which stabilizes in poly-logarithmic rounds w.h.p., albeit being efficient only for some graph families. Meanwhile, Emek et al. [8] devised an algorithm for a simplified version of the Stone Age model that is slightly stronger than the beeping communication model, which stabilizes in $O((D + \log n) \log n)$ rounds w.h.p. on any D-bounded diameter graph, where D is considered a fixed parameter. However, in this context, it would be desirable to relinquish the assumption that vertices possess global information about the network's structure.

Algorithms that do not require any knowledge of the network's topology were also proposed for the beeping model, but they strongly rely on the assumption that, at the beginning of the algorithm, the vertices are in the same fixed initial state, and hence they are not self-stabilizing. One algorithm was proposed by Afek et al. [1] for the full-duplex beeping model, which stabilizes in $O(\log^2 n)$ rounds w.h.p., without requiring vertex knowledge of the

[1] This model is also called the beeping model with collision detection.

network's topology. Later, Jeavons et al. [18] improved this result by proposing an algorithm for the same model, capable of computing an MIS in any n-vertex graph in $O(\log n)$ rounds w.h.p., without requiring any vertex knowledge[2]. Notice that these algorithms are not self-stabilizing because they also rely on the presence of phases of two rounds, implying a synchronization of the vertices modulo two.

1.1 Our Contribution

In this paper, we propose a self-stabilizing algorithm for computing the MIS in the full-duplex beeping model, aiming for a stabilization time of $O(\log n)$ with minimal vertex knowledge about network topology.

We consider the standard fault model, used in most self-stabilizing algorithms [7], where the state of each node is stored in RAM and data in RAM can be corrupted by transients faults (e.g., external events), while the code is stored in ROM and cannot be corrupted. We consider a fault-free execution after a RAM corruption. An algorithm $\mathcal{A}$ is self-stabilizing with termination time T if, after a transient fault within T fault-free steps, it reaches a legal state. This is equivalent to asking that the algorithm $\mathcal{A}$ reaches a legal level after T fault-free steps, starting from an arbitrary state, i.e., without a fixed initialization.

The starting point of our work is Jeavons' algorithm in [18], which is non-self-stabilizing and converges within $O(\log n)$ rounds. We propose two variants that achieve self-stabilization and efficiency across all graph sizes. Our algorithms rely on each vertex's ability to compute a quantity $\ell_{\max}(v)$, which may require access to some information, such as the maximum degree of the graph. The first variant assumes that vertices know an upper bound on the maximum degree Δ and stabilizes in $O(\log n)$ rounds, while the second variant assumes that each vertex knows an upper bound on its own degree and stabilizes in $O(\log n \log \log n)$ rounds. Additionally, we present a third algorithm for the extended beeping model with two channels, stabilizing in $O(\log n)$ time if vertices know an upper bound on the maximum degree among the 1-hop neighborhood. In summary, our contributions yield three algorithms for computing MIS in the beeping model, each highlighting different scenarios based on varying levels of vertex knowledge and beeping channels. Formally, we prove the following theorem.

▶ **Theorem 1.** *Let G be a n-vertex graph.*

1. *If each vertex knows the same upper bound on the maximum degree of G, which is at most polynomial in n, then an MIS can be computed in the beeping model, in a self-stabilizing manner, within $O(\log n)$ rounds w.h.p.*
2. *If each vertex knows an upper bound on its own degree, which is at most polynomial in n, then an MIS can be computed in the beeping model, in a self-stabilizing manner, within $O(\log n \log \log n)$ rounds w.h.p.*
3. *If each vertex knows an upper bound on the maximum degree of all vertices in its 1-hop neighborhood, which is at most polynomial in n, then an MIS can be computed, in the beeping model with two channels, in a self-stabilizing manner, within $O(\log n)$ rounds w.h.p.*

It remains an open question whether a fast, self-stabilizing algorithm computing an MIS in the beeping model can be designed so that no information about the network topology is required to be known by the vertices.

[2] Ghaffari provided a refined analysis for Jeavons at al.'s algorithm in [13].

2 The Algorithm

We assume the full-duplex beeping communication model and the starting point for our algorithm is the beeping, randomized algorithm of Jeavons et al. in [18]. Each vertex v is associated with an adaptive probability $p_t(v)$ of beeping in round t, and the algorithm works in phases, each consisting of two rounds. In the first round of each phase, each vertex v beeps with probability $p_t(v)$ and, if v beeps and all its neighbors are silent, then v joins the MIS. In the second round of each phase, vertices that joined the MIS beep and neighboring vertices hearing a beep become non-MIS vertices. Then, the newly joined MIS and non-MIS vertices remain silent for the rest of the algorithm. The crucial point leading to a $O(\log n)$ global round complexity with high probability, is that active vertices adapt in each phase the beeping probability, initially $p_1(v) = 1/2$ for each vertex v. The value of $p_{t+1}(v)$ is decreased whenever neighboring vertices beep and is increased otherwise. In particular $p_{t+1}(v) = p_t(v)/2$ in the former case and $p_{t+1}(v) = \min\{2p_t(v), 1/2\}$ otherwise. The rationale of this behavior is twofold: to reduce the probability of neighboring vertices attempting to concurrently join the MIS, and to increase the probability of making an attempt to join the MIS in case of no concurrent attempts to do so.

This algorithm is not self-stabilizing for two reasons. First, it works just if at the beginning of the algorithm the probability of beeping of each vertex v is $p_1(v) = 1/2$, and the analysis of the convergence time relies on that. Second, the presence of phases with two rounds requires that the vertices are synchronized modulo two. These reasons are also the main obstacle to making it self-stabilizing. Moreover, in self-stabilizing algorithms, vertices must be able to detect errors, e.g., when a fault forces a vertex to change its state from MIS to non-MIS, and hence stable vertices cannot be silent after they stabilized.

In order to design a self-stabilizing MIS algorithm for the full-duplex beeping model, achieving a $O(\log n)$ global round complexity w.h.p., we dispense with the idea of phases and we change the details of updating the beeping probabilities $p_t(v)$ to overcome the mentioned issues. While keeping the idea of increasing and decreasing the beeping probability depending on whether a beep was received, we refine this behavior in a significant way. As before, when a vertex v beeps while hearing no beeps at the same time it attempts to join the MIS. To signal this to neighboring vertices, vertex v keeps beeping, i.e., it sets its beeping probability $p_t(v)$ to 1. If such a vertex hears a beep in one of the following rounds, it does not immediately give up its attempt to join the MIS, but it keeps beeping with probability 1 for some fixed number rounds. Only after hearing a beep in a certain number of rounds, the vertex changes its behavior back to halving its beeping probability in every round it hears a beep. Furthermore, if the beeping probability decreases over a fixed threshold, the vertex sets its beeping probability to 0 and stops beeping. The complete code is shown in Algorithm 1.

To implement the described behavior, each vertex maintains an integral state variable ℓ, which we call *level*. The value of ℓ for vertex v is in the range $-\ell_{\max}(v), \ldots, \ell_{\max}(v)$, where $\ell_{\max}(v)$ is a fixed value that depends on the vertex's knowledge of some graph parameters. We will see that this value has a strong influence on the analysis of the stabilization time. The value of $\ell_t(v)$ of vertex v in round t implies the beeping probability $p_t(v)$ of v similar to an activation function in an artificial neural network (see Figure 1, in Appendix D). As long as $\ell_t(v) \le 0$ vertex v beeps and $p_t(v) = 1$, if $\ell_t(v) = \ell_{\max}(v)$ it stops beeping and $p_t(v) = 0$, otherwise $p_t(v) = 2^{-\ell_t(v)}$.

In each round t each vertex v updates the value of $\ell_t(v)$ as follows. If v hears a beep then its level increases: $\ell_{t+1}(v) = \min\{\ell_t(v) + 1, \ell_{\max}(v)\}$. Otherwise, $\ell_{t+1}(v) = \max\{\ell_t(v) - 1, 1\}$ unless v was beeping in round t, in this case $\ell_{t+1}(v) = -\ell_{\max}(v)$. Note that the only way the

Algorithm 1 Self-stabilizing version of Jeavons, Scott and Xu's algorithm [18].

```
state: $\ell \in \{-\ell_{\max}(v), \dots, \ell_{\max}(v)\}$
in each round $t = 1, 2, \dots$ do
    if $\ell < \ell_{\max}(v)$ then
        $beep \leftarrow true$ with probability $\min\{2^{-\ell}, 1\}$ and $beep \leftarrow false$ otherwise
    else $beep \leftarrow false$
    if $beep$ then send signal to all neighbors
    receive any signals sent by neighbors
    if any signal received then
        $\ell \leftarrow \min\{\ell + 1, \ell_{\max}(v)\}$
    else if $beep$ then
        $\ell \leftarrow -\ell_{\max}(v)$
    else $\ell \leftarrow \max\{\ell - 1, 1\}$
```

level of a vertex v can decrease below 0 is if v beeps without beeping neighbors. We observe that Algorithm 1 is self-stabilizing if its convergence is guaranteed for every initial value of the levels.

The update rules of the algorithm guarantee that, once the level's value of a vertex v is $-\ell_{\max}(v)$ and each of v's neighbors w has level's value $\ell_{\max}(w)$, then v is such that $p_t(v) = 1$ and all the neighbors u of v are such that $p_t(u) = 0$. This guarantees that v and its neighbors will not change their status as long as no faults occur, and hence they are stable. In this case, v will be a MIS vertex and the neighbors become non-MIS vertices. Also, this strategy allows all vertices to detect faults and react accordingly. But foremost, it allows to determine the stabilization time.

The result and the analysis of the algorithm depend on the values $\ell_{\max}(v)$ of each vertex v, which in turn depends on the knowledge available to each vertex v. We state the detailed results in the following theorems, and notice that we denote with $\deg_2(v) = \max_{u \in N(v) \cup \{v\}} \deg(u)$ the maximum degree in the 1-hop neighborhood of v.

▶ **Theorem 2.** *For any n-vertex graph G, Algorithm 1 computes an MIS, starting from an arbitrary configuration, within $O(\log n)$ rounds w.h.p., provided that $\ell_{\max}(v) = \ell_{\max} \in [\log \Delta + c_1, c_2 \log n]$ for each vertex v and constants $c_1 \geq 15$ and $c_2 > 0$.*

▶ **Theorem 3.** *For any n-vertex graph G, Algorithm 1 computes an MIS, starting from an arbitrary configuration, within $O(\log n \cdot \log \log n)$ rounds w.h.p., provided that $\ell_{\max}(v) \in [2 \log \deg(v) + c_1, c_2 \log n]$ for each vertex v and constants $c_1 \geq 30$ and $c_2 > 0$.*

▶ **Corollary 4.** *There exists a variant of Algorithm 1 for the beeping model with two beeping channels such that, for any n-vertex graph G, it computes an MIS, starting from an arbitrary configuration, within $O(\log n)$ rounds w.h.p., provided that $\ell_{\max}(v) \in [2 \log \deg_2(v) + c_1, c_2 \log n]$, for each vertex v and any constants $c_1 \geq 15$ and $c_2 > 0$.*

To execute Algorithm 1, each vertex v only needs to know the value of $\ell_{max}(v)$. As stated in the three results above, in order to get the time bounds, the value of $\ell_{\max}(v)$ must be in $O(\log n)$ for each v. We remark that to satisfy this requirement it is unnecessary that the value n is known by the vertices. If, for example, $\ell_{max} = \log \Delta + c_1$, then the requirement of Theorem 2 is satisfied, and this only requires each node to know a loose upper bound on Δ.

Roadmap. The rest of the paper is organized as follows. Section 3 contains notations, preliminary definitions, the statement of two key lemmas, Lemmas 8 and 9, and an analysis outline. In Section 4 we give the proof of Theorem 2, and in Section 5 the proof of Theorem 3. The proofs of key Lemmas 8 and 9 can be found in Section 6. The description of the algorithm using two beeping channels and its analysis (the proof of Corollary 4) are deferred to Appendix B. We conclude in Section 7 with a summary and some open problems.

3 Definitions and Analysis Outline

Let $G = (V, E)$ be a graph with n vertices. For each vertex $v \in V$, $N(v)$ denotes the set of v's neighbors in G, and $\deg(v) = |N(v)|$ is the degree of v. Also, $N^+(v) = N(v) \cup \{v\}$ is the set of v's neighbors and v itself. Let $\deg_2(v) = \max_{u \in N^+(v)} \deg(u)$ the maximum degree of all the vertices in $N^+(v)$.

We introduce a few random variables that are used to describe the random process generated by the execution of Algorithm 1. If we denote with $\ell_t(v)$ the level of vertex $v \in V$ at the beginning of round $t \geq 1$, the random execution of the algorithm at time t depends only on the values $\{\ell_t(v)\}_{v \in V}$. We denote with $\mathcal{F}_t$ the filtration of the process until step t, which in particular gives us the values $\{\ell_t(v)\}_{v \in V}$.

We notice that in Algorithm 1 a vertex $v \in V$ is *stable* and permanently added to the MIS prior to round t if $\ell_t(v) = -\ell_{\max}(v)$ and, for all $u \in N(v)$, $\ell_t(u) = \ell_{\max}(u)$. Hence, if we define

$$\mu_t(v) = \min_{u \in N(v)} \frac{\ell_t(u)}{\ell_{\max}(u)},$$

which has value in $[-1, 1]$, we have that the set of vertices that have been added to the final MIS set before round t is defined by

$$I_t = \{v \in V \colon \ell_t(v) = -\ell_{\max}(v) \ \wedge \ \mu_t(v) = 1\}.$$

Moreover, the set of all stable vertices at the beginning of round t consists of the vertices in the MIS and their neighbors, so we define $S_t = I_t \cup N(I_t)$. We notice that the set of stable vertices is increasing in t, i.e., for each $t \geq 1$ we have that $S_t \subseteq S_{t+1}$. For any vertex $v \in V$, we denote with $p_t(v)$ the probability that v beeps during round t, which is

$$p_t(v) = \begin{cases} 1 & \text{if } \ell_t(v) \leq 0 \\ 2^{-\ell_t(v)} & \text{if } 0 < \ell_t(v) < \ell_{\max}(v) \\ 0 & \text{if } \ell_t(v) = \ell_{\max}(v). \end{cases}$$

We also denote with $b_t(v)$ a Bernoulli random variable which takes value 1 if v beeps in round t, i.e., $\mathbf{E}[b_t(v)] = p_t(v)$. We define $B_t(v) = \sum_{u \in N(v)} b_t(u)$ as the number of v's neighbors that beep in round t and $d_t(v) = \mathbf{E}[B_t(v)] = \sum_{u \in N(v)} p_t(u)$ as the expected number of beeping neighbors of v in round t. Note that if $B_t(v) = 0$ then $\mu_t(u) > 0$ for all neighbors u of v. The proof of the following elementary result can be found in Appendix C.

▶ **Lemma 5.** *Let $t > \max_{w \in V} \ell_{\max}(w)$. Then $\ell_t(v) > 0$ or $\mu_t(v) > 0$ for any $v \in V$.*

Lemma 5 implies that in order to prove that our algorithm stabilizes within $O(\log n)$ rounds we can assume that $\ell_t(v) > 0$ or $\mu_t(v) > 0$ for all rounds $t \geq 0$. This is because $\max_{w \in V} \ell_{\max}(w) \in O(\log n)$. Hence, we can ignore the initial $\max_{w \in V} \ell_{\max}(w)$ rounds and start our analysis after those rounds. In particular, $\ell_t(u) \leq 0$ implies $\mu_t(u) > 0$.

We define a vertex to be *prominent* if it has negative or zero level, and a round to be *platinum* for some vertex v if some of v's neighbors is prominent.

▶ **Definition 6** (Prominent Vertices and Platinum Rounds)**.** *A vertex $v \in V$ is* prominent *in round t if $\ell_t(v) \leq 0$. The set of prominent vertices in round t is denoted with PM_t. Moreover, we say that round t is* a platinum round of vertex v *if $N^+(v)$ contains a prominent vertex u, i.e., $u \in N^+(v) \cap PM_t$. We denote with $P_{t,k}(v)$ the number of platinum rounds of vertex v during rounds $\{t, \dots, t+k\}$.*

Clearly, $I_t \subseteq PM_t$. We notice that, since we assume $t > \max_{w \in V} \ell_{\max}(w)$, then Lemma 5 implies that for each platinum round t of v there exists $u \in N^+(v)$ such that $\ell_t(u) \leq 0$ and $\mu_t(u) > 0$, i.e., the probability that none of u's neighbors beeps in round t is positive. Remember, the only possibility for the level of vertex u to become less or equal to 0 is when u beeps while no neighbor of u is beeping. This directly leads to the next lemma.

▶ **Lemma 7.** *If $t > \max_{w \in V} \ell_{\max}(w)$ is a platinum round for vertex v there exists a vertex $u \in N^+(v)$ and a round t' with $t - \ell_{\max}(u) \leq t' \leq t$ in which u was beeping without beeping neighbors and $\ell_{t'+1}(u) = -\ell_{\max}(u)$.*

We define, for any $v \in V$ and $t \geq 1$, the quantities

$$\eta_t(v) = \sum_{u \in N(v) \setminus S_t} 2^{-\ell_{\max}(u)} \quad \text{and} \quad \eta'_t(v) = \sum_{\substack{u \in N(v) \setminus S_t: \\ \ell_{\max}(u) > \ell_{\max}(v)}} 2^{-\ell_{\max}(v)}.$$

For the moment, the definitions of $\eta_t(v)$ and $\eta'_t(v)$ are rather technical, but they will be used to upper bound the value of $d_{t+1}(v)$. We notice that $\eta_t(v)$ and $\eta'_t(v)$ are both decreasing in t, since $S_t \subseteq S_{t+1}$.

The following two lemmas are the key to prove Theorems 2 and 3 and Corollary 4, their proofs are deferred to Section 6. For a fixed $v \in V$ the next lemma tells us how many rounds we have to wait in order to have a platinum round of v.

▶ **Lemma 8** (Lower Bound on Platinum Rounds)**.** *Assume that $\ell_{\max}(w) \geq \log \deg(w) + 4$ for all $w \in V$. Consider a vertex $v \in V$ and a round $t > \max_{w \in V} \ell_{\max}(w)$ such that t is not a platinum round of v, and $\eta_t(v) \leq 0.0001$. Let $\tau^{(v)}(t) = min\{m \geq 0 : P_{t,m}(v) \geq 1\}$. Then*

$$\mathbf{Pr}\left[\tau^{(v)}(t) \geq k \mid \mathcal{F}_t\right] \leq e^{-\gamma k},$$

for $\gamma = e^{-30}$ and any $k \geq 2\gamma^{-1}\ell_{\max}(v)$.

We notice that, if $\ell_{\max}(w)$ is constant over all vertices $w \in V$, i.e., $\ell_{\max}(w) = \ell_{\max}$ for every vertex $w \in V$, then the existence of a platinum round t of v such that $t > \ell_{\max}$ is by Lemma 7 sufficient to guarantee that v will be stable at the latest in round $t + \ell_{\max}$. Indeed, Lemma 7 implies the existence of a round $1 \leq t' \leq t$ and a vertex $u \in N^+(v)$ such that u was beeping in round t' without beeping neighbors, and so $\ell_{t'+1}(u) = -\ell_{\max}$ and $\mu_{t'+1}(u) > 0$. This implies that u beeps in the following $\ell_{\max}$ rounds, during which all neighbors of u will increase their level until they reach maximum level $\ell_{\max}$. This implies that u is such that $\ell_{t+\ell_{\max}}(u) \leq 0$ and $\mu_{t+\ell_{\max}}(u) = 1$, and hence $u, v \in S_{t+\ell_{\max}}$ and $u \in I_{t+\ell_{\max}}$.

However, when $\ell_{\max}(w)$ is not constant, the analysis becomes considerably more complicated, since the existence of a platinum round of v does not necessarily imply the subsequent stabilization of v. Consider now some round $t > \max_{w \in V} \ell_{\max}(w)$ which is platinum for v, and let $u \in N^+(v)$ be a prominent vertex. After round t, two things may happen:

(i) In some round $t + m$ with $m \geq 1$, u is no longer prominent, and hence $u \notin I_{t+m}$ and u, v may not be stable in round $t + m$;

(ii) In some round $t + m$ with $m \geq 0$, vertex u is prominent and all its neighbors have reached the maximum level, i.e., $\mu_{t+m}(u) = 1$, and so $u \in I_{t+m}$ and $u, v \in S_{t+m}$.

In the next lemma, we characterize the distribution of rounds t for the above two cases. Let

$$\sigma^{(u)}_{\text{out}}(t) = \min\{m \geq 0 : u \notin PM_{t+m}\}$$
$$\sigma^{(u)}_{\text{in}}(t) = \min\{m \geq 0 : u \in I_{t+m}\}$$
$$\sigma^{(u)}(t) = \min\{\sigma^{(u)}_{\text{out}}(t),\ \sigma^{(u)}_{\text{in}}(t)\}.$$

▶ **Lemma 9** (Stopping Times for Platinum Rounds)**.** *Assume that $\ell_{\max}(w) \geq \log \deg(w) + 4$ for all $w \in V$. Consider a round $t > \max_{w \in V} \ell_{\max}(w)$ and a vertex $u \in PM_t \setminus S_t$. Then*

(a) $\mathbf{Pr}\left[\sigma^{(u)}(t) = \sigma^{(u)}_{in}(t) \wedge \sigma^{(u)}(t) < \max_{w \in N(u)} \ell_{\max}(w) \mid \mathcal{F}_t\right] \geq 3^{-\eta'_t(u)}$*;*

(b) $\mathbf{Pr}\left[\sigma^{(u)}(t) = \sigma_{out}(t) \wedge \sigma^{(u)}(t) > \ell_{\max}(u) + x \mid \mathcal{F}_t\right] \leq \eta'_t(u) 2^{-x}$ *for any $x \geq 0$.*

3.1 Analysis Overview

We first give an overview of the proofs of Lemmas 8 and 9, and then we will see how to use these results to prove Theorems 2 and 3 and Corollary 4. The proof of Lemma 8 has as a starting point the proof in [13], but then it develops differently. First, as in [13], we define a further type of round called *golden round*, which are rounds having constant probability of becoming platinum in the subsequent round. We prove that, for any vertex v in any fixed interval of rounds of length $k = \Omega(\ell_{\max}(v))$, we have a constant fraction of golden rounds with probability at least $1 - e^{-\Omega(k)}$, conditioned on the absence of platinum rounds during that time interval. To prove the latter, as in [13], we analyze the development of the function $d_t(v)$ – the expected number of beeping neighbors of v in round t – during this time span. Note that platinum rounds and the conditioning were not considered in [13] and are essential in our proof and setting.

The proof of Lemma 9 relies on Lemma 7. Assuming that u is prominent at time t, we characterize the probabilities with which, after round t, u reaches again a positive level or stabilizes. From Lemma 7 there exists a round $t - \ell_{\max}(u) \leq t' \leq t$ where we have that $\ell_{t'+1}(u) = -\ell_{\max}(u)$ and then trivially $d_{t'+1}(u) = \sum_{w \in N(u)} p_{t'+1}(u) \leq \deg(u)$. Then, in the subsequent $\ell_{\max}(u)$ rounds, vertex u keeps beeping regardless the behavior of the vertices in $N(u)$. Hence, $\ell_{t'+1+\ell_{\max}(u)}(w) = \min\{\ell_{\max}(w), \ell_{t'+1}(w) + \ell_{\max}(u)\}$ for each $w \in N(u)$ and thus, $p_{t'+1+\ell_{\max}(u)}(w) \leq 2^{-\ell_{\max}(u)}$ if $\ell_{t'+1+\ell_{\max}(u)}(w) \neq \ell_{\max}(w)$. This implies that

$$d_{t'+1+\ell_{\max}(u)}(u) = \sum_{w \in N(u)} p_{t'+1+\ell_{\max}(u)}(w) \leq \sum_{\substack{w \in N(u) \setminus S_{t'+1}: \\ \ell_{\max}(w) > \ell_{\max}(u)}} 2^{-\ell_{\max}(u)} \leq \eta'_{t'+1}(u).$$

We will see that this implies that the vertices in $N(u)$ will reach their maximal level with probability at least $3^{-\eta'_{t+1}(u)}$, and so in this case the platinum round leads to the stabilization of u. On the other hand, part (b) of the lemma follows from the observation that, after the first $\ell_{\max}(u)$ rounds after t, the probability that some vertex in $N(u)$ beeps decreases in each round by a constant factor.

Theorem 2 and Corollary 4 follow from the observation, already stated above, that if $\ell_{\max}(w)$ is constant over w then, for each vertex v, one platinum round is sufficient to guarantee the stabilization of v. Moreover, the choices of $\ell_{\max}(w)$ specified in the theorems guarantee that $\eta_t(v) \leq 0.0001$ for every v and $t \geq 1$, and so Lemma 8 can always be used for each non-platinum round t, and implies that we have to wait at most $O(\log n)$ rounds to have a platinum round for each vertex v w.h.p., that in turns imply stabilization.

The proof of Theorem 3 is considerably harder. In this case, we can have several sequences of consecutive platinum rounds, intermittent by sequences of consecutive non-platinum rounds, until we reach a platinum round leading to the stabilization of the vertex. The analysis relies on two main parts:

(1) We split the vertices in $O(\log\log n)$ sets V_i. Before analyzing the stabilization of a vertex $v \in V_i$, we wait for round T_i in which all vertices in $\cup_{j<i} V_j$ have stabilized. The sets V_i are defined according to the values $\ell_{\max}(v)$ of the vertices. According to the definition of T_i, we can apply, for each round $t \geq T_i$, Lemmas 8 and 9 to vertices in V_i.

(2) We then prove that, after round T_i, each vertex $v \in V_i$ stabilize in $O(\log n)$ additional rounds w.h.p. The analysis of the latter statement relies on Lemmas 8 and 9, which characterize the lengths of three times intervals: that of the non-platinum rounds, of the platinum rounds, and that of the number of platinum rounds not leading to the stabilization of vertex v.

4 Knowledge of Maximum Degree Δ (Proof of Theorem 2)

The following proof is a warm-up for the general case. It is directly implied by Lemma 8 and the choice of $\ell_{\max}(v)$.

Proof of Theorem 2. As already mentioned, since $\ell_{\max}(v)$ is defined independently of v, each vertex v just requires a single platinum round to become stable in at most $\ell_{\max}$ rounds. Indeed, for each $v \in V$ and each $t \geq 1$,

$$\eta_t(v) \leq \sum_{u \in N(v)} 2^{-\log \Delta - 15} \leq 2^{-15} \leq 0.0001 \quad \text{and} \quad \eta'_t(v) = 0.$$

This implies that, if $t = 2\ell_{\max}$ and $v \in V$, we have $\eta_t(v) \leq 0.0001$. Hence, by Lemma 8, if we take $m = 2\gamma^{-1} \log n$ (where γ is defined in Lemma 8), we have that $\mathbf{Pr}\left[\tau^{(v)}(t) \leq m \mid \mathcal{F}_t\right] \geq 1 - 1/n^2$, and so $P_{t,m}(v) \geq 1$ with probability at least $1 - 1/n^2$. Then, from Lemma 9(a), and since $\eta'_{t+m}(v) = 0$, we have that, given $\mathcal{F}_{m+t}$, the vertex v is stable after at most $\ell_{\max}$ rounds with probability 1. Hence, vertex v is stable with probability $1 - 1/n^2$ after $t + m + \ell_{\max}$ rounds, and since $\ell_{\max} = O(\log n)$ we have that $t + m + \ell_{\max} = O(\log n)$. The theorem follows from the union bound over all the vertices. ◀

5 Knowledge of Own Degree (Proof of Theorem 3)

In this section we prove Theorem 3. First, we prove the following lemma.

▶ **Lemma 10.** *Assume that $\ell_{\max}(w) \geq 2 \log \deg(w)$ for every $w \in V$ and that, for some $c = O(1)$, $\max_{w \in V} \ell_{\max}(w) \leq c \log n$. Consider a vertex $v \in V$ and a round $t > \max_{w \in V} \ell_{\max}(w)$ such that $\eta_t(v) \leq 0.0001$ and $\ell_{\max}(v) \leq 2\ell_{\max}(u)$ for each $u \in N(v) \setminus S_t$. Then, there exists a constant $M = O(1)$ such that $\mathbf{Pr}\left[v \in S_{t+m} \mid \mathcal{F}_t\right] \geq 1 - 1/n^2$, provided $m = M \log n$.*

Proof. We fix the execution up to the end of round t, so we do not have to condition probabilities on $\mathcal{F}_t$. We consider the sequence of rounds (which may also be infinite, with $J = +\infty$)

$$t \leq t + \tau_1 \leq t + \tau_1 + \sigma_1 = m_1 + t \leq \cdots \leq t + m_{J-1} + \tau_J \leq t + m_{J-1} + \tau_J + \sigma_J = t + m_J,$$

and the corresponding sequence of vertices $v_1, v_2, \ldots, v_J \in N^+(v) \setminus S_t$ such that

1. $t + m_{i-1} + \tau_i$ is platinum for v and $v_i \in PM_{t+m_{i-1}+\tau_i} \cap N^+(v)$ for each $i = 1, \ldots, J$;
2. $m_i = m_{i-1} + \tau_i + \sigma_i$ is such that $v_i \notin PM_{t+m_i}$ for each $i = 1, \ldots J - 1$;
3. $J = \min\{h \geq 1 : v_h \in I_{m_h+t}\}$, hence $v_J \in I_{t+m_J}$ and $v \in S_{t+m_J}$. If v never stabilizes, we define $J = +\infty$ and the sequence $v_1, v_2, \ldots$ has infinite length.

We observe that σ_i and τ_i are defined such that

$$\tau_i = \tau^{(v_i)}(t+m_{i-1}), \quad \sigma_i = \sigma^{(v_i)}(t+m_{i-1}+\tau_i) \quad \text{and} \quad \sigma_J = \sigma_{\text{in}}^{(v_J)}(t+m_{J-1}+\tau_J). \tag{1}$$

Consider the following two facts:

(i) $\sum_{i=1}^{J}(\sigma_i + \ell_{\max}(v_i)) \le M_1 \log n$ for some $M_1 = \Theta(1)$ with probability at least $1 - 1/n^3$;

(ii) Provided that $\sum_{i=1}^{J}(\sigma_i + \ell_{\max}(v_i)) \le M_1 \log n$, it holds $\sum_{i=1}^{J} \tau_i \le M_2 \log n$ for some $M_2 = \Theta(1)$ with probability at least $1 - 1/n^3$.

The above facts (i) and (ii) prove the lemma. Indeed, if $m = M_1 \log n + M_2 \log n$, we have that

$$\begin{aligned}
&\mathbf{Pr}\left[v \notin S_{t+m}\right] \le \mathbf{Pr}\left[\sum_{i=1}^{J}(\sigma_i+\tau_i) \ge m\right] \le \mathbf{Pr}\left[\sum_{i=1}^{J}\sigma_i \ge M_1 \log n \vee \sum_{i=1}^{J}\tau_i \ge M_2 \log n\right] \\
&\le \mathbf{Pr}\left[\sum_{i=1}^{J}\tau_i \ge M_2 \log n \;\middle|\; \sum_{i=1}^{J}(\sigma_i + \ell_{\max}(v_i)) \le M_1 \log n\right] + \\
&\quad + \mathbf{Pr}\left[\sum_{i=1}^{J}(\sigma_i + \ell_{\max}(v_i)) \ge M_1 \log n\right] \le \frac{2}{n^3}.
\end{aligned}$$

Now we prove (i) and (ii) separately.

Proof of (i). We remark that, in this first step, we are just looking at the randomness of the execution during the time intervals $[t+\tau_i+1, t+\tau_i+\sigma_i]$ for $i = 1, \ldots, J$. We notice that

$$\begin{aligned}
&\mathbf{Pr}\left[\sum_{i=1}^{J}\sigma_i + \ell_{\max}(v_i) \ge M_1 \log n\right] \le \mathbf{Pr}\left[\sum_{i=1}^{J}\ell_{\max}(v_i) \ge 7\log n \vee \sigma_J \ge \max_{w\in V}\ell_{\max}(w)\right] \\
&+ \mathbf{Pr}\left[\sum_{i=1}^{J}\ell_{\max}(v_i) \le 7\log n \wedge \sigma_J \le \max_{w\in V}\ell_{\max}(w) \wedge \sum_{i=1}^{J}(\sigma_i + \ell_{\max}(v_i)) \ge M_1 \log n\right]
\end{aligned} \tag{2}$$

We start by showing that the first term in the inequality above is at most $1/(2n^3)$. Let $h = \sup\{j \ge 1 : \sum_{i=1}^{j} \ell_{\max}(v_i) \ge 7\log n\}$ and notice that, since $\min_{v\in V}\ell_{\max}(v) \ge 1$, from the minimality of h we have that $h \le 7\log n$. Assume that $h \le J$, otherwise the inequality follows trivially. Lemma 9(a) together with (1) yields

$$\begin{aligned}
&\mathbf{Pr}\left[\sum_{i=1}^{J}\ell_{\max}(v_i) \ge 7\log n \vee \sigma_J \ge \max_{w\in V}\ell_{\max}(w)\right] \\
&\le \mathbf{Pr}\left[\bigcap_{i=1}^{h}\{v_i \notin PM_{t_{i+1}} \vee \sigma_i \ge \max_{w\in V}\ell_{\max}(w)\}\right] \le \prod_{i=1}^{h}\left(1 - 3^{-\eta'_t(v_i)}\right) \le 2\prod_{i=1}^{h}\eta'_t(v_i).
\end{aligned} \tag{3}$$

Moreover, we have that

$$\begin{aligned}
\eta'_t(v_i) &\le \sum_{\substack{w\in N(v_i):\\ \ell_{\max}(w)>\ell_{\max}(v_i)}} 2^{-\ell_{\max}(v_i)} \le \deg(v_i)\cdot 2^{-\ell_{\max}(v_i)} \\
&\le \frac{\deg(v_i)}{2^{\ell_{\max}(v_i)/2}} 2^{-\ell_{\max}(v_i)/2} \le 2^{-\ell_{\max}(v_i)/2},
\end{aligned}$$

where the last inequality follows from the fact that $\ell_{\max}(v_i) \geq 2\log\deg(v_i)$. Hence, from the latter inequality and (2) we have that

$$\mathbf{Pr}\left[\sum_{i=1}^{J}\ell_{\max}(v_i) \geq 7\log n \vee \sigma_J \geq \max_{w\in V}\ell_{\max}(w)\right] \leq 2\prod_{i=1}^{h}\eta'_t(v_i) \leq 2^{-\sum_{i=1}^{h}\ell_{\max}(v_i)/2+1} \leq \frac{1}{2n^3},$$

where the last inequality follows from the fact that $\sum_{i=1}^{h}\ell_{\max}(v_i) \geq 7\log n$.

We proceed by showing that the term in (2) is bounded by $1/(2n^3)$. From (1) and Lemma 9(b) we have that, for each $i = 1,\dots,J-1$, the random variables σ_i are stochastically dominated by $\ell_{\max}(v_i) + Y_i$, where Y_i are independent geometric random variables with parameter $1/2$. We have that, fixing $M_1 = 36 + c$ and since $\max_{w\in V}\ell_{\max}(w) \leq c\log n$,

$$\begin{aligned}
&\mathbf{Pr}\left[\sum_{i=1}^{J}\ell_{\max}(v_i) \leq 7\log n \wedge \sigma_J \leq \max_{w\in V}\ell_{\max}(w) \wedge \sum_{i=1}^{J}\sigma_i + \ell_{\max}(v_i) \geq M_1\log n\right]\\
&\leq \mathbf{Pr}\left[\sum_{i=1}^{J-1}(Y_i + 2\ell_{\max}(v_i)) + \sigma_J \geq M_1\log n \wedge \sum_{i=1}^{J}\ell_{\max}(v_i) \leq 7\log n \wedge \sigma_J \leq \max_{w\in V}\ell_{\max}(w)\right]\\
&\leq \mathbf{Pr}\left[\sum_{i=1}^{J-1}Y_i \geq 2J + 8\log n \wedge J \leq 7\log n\right]\\
&= \mathbf{Pr}\left[\mathrm{Bin}(2J + 8\log n, \tfrac{1}{2}) \leq J - 1 \wedge J \leq 7\log n\right] \leq \frac{1}{2n^3},
\end{aligned}$$

where the last inequality follows from Lemma 16, in Appendix A, and since $\sum_{i=1}^{J}\ell_{\max}(v_i) \leq 7\log n$ implies that $J \leq 7\log n$.

Proof of (ii). This time we are looking at the randomness of the rounds $[t + m_i + 1, t + m_i + \tau_{i+1}]$ for $i = 1,\dots,J$. From (1) and Lemma 8, we have that the random variables τ_i are stochastically dominated by $2\gamma^{-1}\ell_{\max}(v_i) + X_i$, where X_i are i.i.d. geometric random variable with parameter $p = 1 - e^{-\gamma}$, where $\gamma = e^{-30}$. Then, we have that, assuming that $\sum_{i=1}^{J}(\sigma_i + \ell_{\max}(v_i)) \leq M_1\log n$ and in particular that $\sum_{i=1}^{J}\ell_{\max}(v_i) \leq M_1\log n$, if $M_2 = 2\gamma^{-1}M_1 + M_1/p + 4/p^2$, then

$$\begin{aligned}
&\mathbf{Pr}\left[\sum_{i=1}^{J}\tau_i \geq M_2\log n \mid \sum_{i=1}^{J}\ell_{\max}(v_i) \leq M_1\log n\right]\\
&\leq \mathbf{Pr}\left[\sum_{i=1}^{J}X_i + 2\gamma^{-1}\ell_{\max}(v_i) \geq M_2\log n \mid \sum_{i=1}^{J}\ell_{\max}(v_i) \leq M_1\log n\right]\\
&\leq \mathbf{Pr}\left[\sum_{i=1}^{J}X_i \geq \tfrac{J}{p} + \tfrac{4\log n}{p^2} \mid \sum_{i=1}^{J}\ell_{\max}(v_i) \leq M_1\log n\right]\\
&= \mathbf{Pr}\left[\mathrm{Bin}(\tfrac{J}{p} + \tfrac{4\log n}{p^2}, p) \leq J \mid \sum_{i=1}^{J}\ell_{\max}(v_i) \leq M_1\log n\right] \leq \frac{1}{n^3} \qquad (4)
\end{aligned}$$

where (4) follows from Lemma 16, and the last inequality follows from the fact that $p = 1 - e^{-\gamma}$ and that $\sum_{i=1}^{J}\ell_{\max}(v_i) \leq M_1\log n$ implies that $J \leq M_1\log n$. ◀

We now can proceed with the proof of Theorem 3.

Proof of Theorem 3. We have $2\log\deg(w) + 30 \leq \ell_{\max}(w) \leq c_2\log n$ for every $w \in V$ and some $c_2 = O(1)$. For each $i = 1,\dots,c_2\log\log n$, we consider the following subsets of vertices

$$V_i = \{v \in V : \ell_{\max}(v) \in [2^i, 2^{i+1}]\}.$$

Let T_i be the round until all the vertices in $\cup_{j\le i}V_j$ are stabilized, i.e.,

$$T_i = \min\{t' \ge 1 : \cup_{j\le i}V_j \subseteq S_{t'}\}.$$

We have that, for each vertex $v \in V_{i+1}$ and each $t \ge T_i$,

$$2\ell_{\max}(u) \ge \ell_{\max}(v) \quad \forall u \in N^+(v) \setminus S_t.$$

Indeed, we have $u \notin \cup_{j\le i}V_j$ since $u \notin S_t$ and $t \ge T_i$. Hence, $\ell_{\max}(u) \ge 2^{i+1}$. Since $v \in V_{i+1}$, $\ell_{\max}(v) \le 2^{i+2}$ and so $2\ell_{\max}(u) \ge \ell_{\max}(v)$. We also have, for each $t \ge T_i$ and each $v \in V_{i+1}$

$$\eta_t(v) \le \sum_{u\in N(v)\setminus S_t} 2^{-\ell_{\max}(u)} \le \sum_{u\in N(v)\setminus S_t} 2^{-\ell_{\max}(v)/2} \le \sum_{u\in N(v)\setminus S_t} \frac{1}{\deg(v)}2^{-15} \le 0.0001,$$

where the second inequality follows from the fact that $2\ell_{\max}(u) \ge \ell_{\max}(v)$, and the third inequality since $\ell_{\max}(v) \ge 2\log\deg(v) + 30$.

We can now apply Lemma 10, if $t \ge \max\{T_i, \max_{w\in V}\ell_{\max}(w) + 1\}$, to all the vertices $v \in V_{i+1}$, obtaining (with an union bound over all the vertices in V_i) the existence of a round $m_i = O(\log n)$ such that $\mathbf{Pr}\left[V_{i+1} \subseteq S_{t+m_i}\right] \ge 1 - 1/n$. Applying this argument iteratively for each $i = 1, \dots, c_2 \log\log n$, we obtain the existence of a round

$$m = \sum_{i=1}^{\log\log n} m_i = O(\log n \cdot \log\log n)$$

such that all vertices are stable w.h.p. at round m. ◀

6 Proof of Key Lemmas

6.1 Lower Bound on Platinum Rounds (Proof of Lemma 8)

Before proving Lemma 8, we introduce some definitions and preliminary lemmas.

▶ **Definition 11** (Light Vertices). *A vertex $v \in V$ is called* light *in round t if $\mu_t(v) > 0 \wedge (d_t(v) \le 10 \vee \ell_t(v) \le 0)$. We denote with L_t the set of light vertices at round t and with $H_t = V \setminus L_t$ the set of* heavy *(non-light) vertices at round t.*

Intuitively, a light vertex v is prominent or has a positive, constant probability of not receiving a beep signal during round t and, in the latter case, if $p_t(v)$ is large enough, v has a constant probability of beeping without beeping neighbors during round t. We remark that the condition $\mu_t(v) > 0$ is necessary since, if $\mu_t(v) = 0$, the vertex v hears a beep during round t with probability 1.

We denote with $d_t^L(v) = \sum_{u\in N(v)\cap L_t} p_t(u)$ the expected number of beeping light neighbors of v in round t, and with $d_t^H(v) = \sum_{u\in N(v)\cap H_t} p_t(u)$ the expected number of beeping heavy neighbors of v in round t. We notice that $d_t(v) = d_t^L(v) + d_t^H(v)$.

▶ **Definition 12** (Golden Rounds). *Round t is a* golden round *of vertex v if one of the following two conditions are satisfied:*

(a) *$\ell_t(v) \le 1$ and $d_t(v) \le 0.02$;*
(b) *$d_t^L(v) > 0.001$.*

We denote with $G_{t,k}(v)$ the number of golden rounds of vertex v during rounds $\{t, \dots, t+k\}$.

In the next section, we will give a lower bound on the number of golden rounds.

6.1.1 Lower Bound on Golden Rounds

▶ **Lemma 13.** *Assume that $\ell_{\max}(w) \geq \log \deg(w) + 4$ for all $w \in V$. Consider a vertex $v \in V$ and a round $t > \max_{w \in V} \ell_{\max}(w)$ such that t is not a platinum round of v, and $\eta_t(v) \leq 0.0001$. Let $\tau^{(v)}(t)$ be defined as in Lemma 8. Then, we have that, for any $k \geq 70 \cdot \ell_{\max}(v)$,*

$$\mathbf{Pr}\left[G_{t,k}(v) \leq 0.05k \wedge \tau^{(v)}(t) > k \mid \mathcal{F}_t\right] \leq e^{-k/100}.$$

We notice that, if round t is not a platinum round of v, every round $s \in [t, \tau^{(v)}(t)]$ is also not a platinum round of v, since the only way a vertex in $N^+(v)$ can take a negative level is by beeping without beeping neighbors, and $\tau^{(v)}(t)$ is the first round that happens. The proof of Lemma 13, which is deferred to Appendix C, relies on the following result.

▶ **Lemma 14.** *Let $v \in V$ and $t > \max_{w \in V} \ell_{\max}(w)$ such that round t is not a platinum round of v and $\eta_t(v) \leq 0.0001$.*
(a) *If $d_t(v) \leq 0.02$, then $\ell_{t+1}(v) \leq \max\{1, \ell_t(v) - 1\}$ with probability at least 0.97.*
(b) *If $d_t(v) > 0.01$ and $d_t^L(v) < 0.01 d_t(v)$, then with probability at least 0.97, we have that $d_{t+1}(v) < 0.6 d_t(v)$ or that $t+1$ is a platinum round for v.*

Proof. We fix the execution up to the end of round t, so we do not have to condition probabilities on $\mathcal{F}_t$. In what follows, we prove separately the two statements.

We prove (a) first. Since $d_t(v) = \sum_{u \in N(v)} p_t(u) \leq 0.02$ it follows that $p_t(u) \leq \frac{1}{2}$ for all $u \in N(v)$. Thus, the probability that no neighbor of v beeps is at least $\prod_{u \in N(v)} (1 - p_t(u)) \geq 4^{-d_t(v)} \geq 0.97$. Hence, $\mathbf{Pr}[\ell_{t+1}(v) \leq \max\{\ell_t(v) - 1, 1\}] \geq 0.97$.

Next we prove (b). Since t is not a platinum round of v, we have that for each $u \in N^+(v)$, $\ell_t(u) \geq 1$. Moreover, we notice that there may be in round t a beeping vertex $u \in N^+(v)$ that does not receive a signal, and so $\ell_{t+1}(u) = -\ell_{\max}(u) \leq 0$.

For any vertex $u \in N^+(v)$, we have the following upper bounds for $p_{t+1}(u)$ (recall that $\ell_t(u) > 0$ since t is not a platinum round of v):

$$p_{t+1}(u) \leq \begin{cases} 2^{-\ell_{\max}(u)+1} & \text{if } \ell_t(u) = \ell_{\max}(u) \text{ and } u \notin S_t \\ 0 & \text{if } \ell_t(u) = \ell_{\max}(u) \text{ and } u \in S_t \\ \frac{p_t(u)}{2} & \text{if } B_t(u) \geq 1 \text{ and } \ell_t(u) \neq \ell_{\max}(u) \\ 2p_t(u) & \text{if } B_t(u) = b_t(u) = 0 \text{ and } \ell_t(u) \neq \ell_{\max}(u) \\ 1 & \text{if } B_t(u) = 0, b_t(u) = 1 \text{ and } \ell_t(u) \neq \ell_{\max}(u) \end{cases}$$

The last case, i.e., when $B_t(u) = 0$ and $b_t(u) = 1$ implies that $t+1$ is a platinum round for v, and that $\ell_{t+1}(u) = -\ell_{\max}(u)$. Define $J_{t+1}(v)$ the set of such vertices, i.e., the set of vertices in $N(v)$ beeping in round t without beeping neighbors. Then,

$$\begin{aligned} d_{t+1}(v) &\leq \sum_{\substack{u \in N(v) \setminus S_t: \\ \ell_t(u) = \ell_{\max}(u)}} 2^{-\ell_{\max}(u)+1} + \sum_{\substack{u \in N(v): \\ B_t(u) \geq 1 \\ \ell_t(u) \neq \ell_{\max}(u)}} \frac{p_t(u)}{2} + \sum_{\substack{u \in N(v): \\ B_t(u) = b_t(u) = 0}} 2p_t(u) + J_{t+1}(v) \\ &\leq 2\eta_t(v) + \sum_{u \in N(v) \cap H_t} p_t(u) \left(\frac{1}{2} + 2 \cdot \mathbb{1}_{B_t(u)=0}\right) + \sum_{u \in N(v) \cap L_t} 2p_t(u) + J_{t+1}(v). \end{aligned}$$

We notice that, since $d_t^L(v) = \sum_{u \in N(v) \cap L_t} p_t(u) \leq 0.01 d_t(v)$ and $\eta_t(v) \leq 0.001$, we have that

$$d_{t+1}(v) \leq 0.0002 + 0.02 d_t(v) + J_{t+1}(v) + \sum_{u \in N(v) \cap H_t} p_t(u) \left(\frac{1}{2} + 2 \cdot \mathbb{1}_{B_t(u)=0}\right).$$

We want to bound, for each $u \in N(v) \cap H_t$, the probability that $B_t(u) = 0$. Since $u \in N(v) \cap H_t$ and $\ell_t(u) \geq 1$, then $d_t(u) \geq 10$ or $\mu_t(u) = 0$. In the latter case, we know that u has some neighbor $u' \in N(u)$ with $p_t(u') = 1$. Hence, we have that $\mathbf{Pr}\left[B_t(u) = 0\right] = 0$. In the former case, we have that none of u's neighbors is beeping with probability at most

$$\prod_{w \in N(u)} (1 - p_t(w)) \leq e^{-d_t(u)} \leq e^{-10}.$$

Hence, we have that, for each $u \in N(v) \cap H_t$, $\mathbf{Pr}\left[B_t(u) = 0\right] \leq e^{-10}$. So,

$$\mathbf{E}\left[\sum_{u \in N(v) \cap H_t} 2p_t(u) \mathbb{1}_{B_t(u)=0}\right] \leq \sum_{u \in N(v) \cap H_t} 2p_t(u) e^{-10}.$$

Markov's inequality implies that $\sum_{u \in N(v) \cap H_t} 2p_t(u) \mathbb{1}_{B_t(u)=0} \leq 0.01 \sum_{u \in N(v) \cap H_t} 2p_t(u)$ with probability at least $1 - \frac{e^{-10}}{0.01} \geq 0.97$. Thus, with probability at least 0.97, we have that

$$d_{t+1}(v) \leq 0.0002 + 0.02 d_t(v) + J_{t+1}(v) + 0.5 d_t(v) + 0.02 d_t(v) \leq 0.6 d_t(v) + J_{t+1}(v), \quad (5)$$

where the last inequality follows by noticing that $d_t(v) > 0.01$ and hence $0.0002 < 0.02 d_t(v)$. This yields the lemma, since (5) implies that either $d_{t+1}(v) < 0.6 d_t(v)$, or $J_{t+1}(v) > 0$, and hence $t+1$ is a platinum round for v. ◀

6.1.2 From Golden to Platinum Rounds

We first notice that, for each golden round t of v, round $t+1$ is platinum for v with constant probability. Indeed, we have the following lemma.

▶ **Lemma 15.** *Let $t > \max_{w \in V} \ell_{\max}(w)$ be a non-platinum round of v, and consider $\tau^{(v)}(t)$ as in Lemma 8. Then, we have that, for each $t \leq s < \tau^{(v)}(t)$ which is golden for v, $\mathbf{Pr}\left[\tau^{(v)}(t) = s+1 \mid \mathcal{F}_s\right] \geq \gamma$, where $\gamma \geq e^{-27}$.*

Proof. Since $t \leq s < \tau^{(v)}(t)$, s is not a platinum round of v, every vertex $u \in N^+(V)$ is such that $\ell_s(v) \geq 1$. In what follows, we prove that, with constant probability $\gamma > 0$, during round s, there is a vertex $u \in N^+(v)$ such that $B_s(u) = 0$ and $b_s(u) = 1$. This implies that $\ell_{s+1}(u) = -\ell_{\max}(u)$ and that $\mu_{s+1}(u) > 0$, hence that $s+1$ is platinum for v. Since s is golden for v, we have that part (a) or (b) of Definition 12 holds.

First, assume that (a) holds, thus $\ell_s(v) \leq 1$ and $d_s(v) \leq 0.02$. In this case, with probability at least 0.48, we have that $B_s(v) = 0$ and $b_s(v) = 1$ and so $s+1$ is platinum for v. Indeed, the expected number of beeping neighbors of v during round s is $d_s(v) \leq 0.02$. Therefore, for Markov's inequality, $\mathbf{Pr}\left[B_s(v) \geq 1\right] \leq 0.02$, and v beeps with probability at least $1/2$, and then the level of v is updated to zero with probability at least $\frac{1}{2} \cdot 0.98 > 0.48$.

We now assume that round s satisfies (b), therefore that $d_s^L(v) \geq 0.001$. We will prove that, in such types of rounds, with probability at least a constant γ, there is a beeping vertex $u \in N(v)$ with no beeping neighbors during round s. Let $k = |N(v) \cap L_s|$ be the number of light neighbors of v, and denote $\{w_1, \ldots, w_k\} = N(v) \cap L_s$. We remark that all light vertices w_i are such that $\ell_s(u) > 0$ for each $u \in N(w_i)$ and $d_s(w_i) \leq 10$. We define $\mathcal{E}_i$ to be the event indicating that vertex w_i is beeping during round s. Let $\mathcal{E} = \cup_i \mathcal{E}_i$. We have that,

$$\Pr\left[\mathcal{E}\right] \geq 1 - \prod_{w \in N(v) \cap L_s} (1 - p_t(w)) \geq 1 - e^{-\sum_{w \in N(v) \cap L_s} p_s(w)} = 1 - e^{-d_s^L(w)} \geq 1 - e^{-0.001}.$$

Suppose that $\mathcal{E}$ occurs, and let j be the smallest index such that $\mathcal{E}_j$ occurs, i.e., $\bar{\mathcal{E}}_1 \cap \bar{\mathcal{E}}_2 \cap \cdots \cap \bar{\mathcal{E}}_{j-1} \cap \mathcal{E}_j$ occurs. If $G_j = N(w_j) \setminus \{w_1, \ldots, w_{j-1}\}$, then the probability that no neighbor of w_j in G_j beeps is at least

$$\prod_{u \in G_j} (1 - p_s(u)) \geq \prod_{u \in N(w_j)} (1 - p_s(u)) \geq e^{-2d_s(w)} \geq e^{-20}.$$

where the first inequality follows from the fact that, since w_j is light, $\mu_s(w_j) > 0$ and so each $u \in N(w_j)$ is such that $\ell_s(u) \geq 1$. Combining this with the previous inequality, we have that a vertex $w \in N(v)$ with $d_s(w) \leq 10$ is beeping with no beeping neighbors with probability at least $e^{-20}(1 - e^{-0.001}) > e^{-27} = \gamma$. ◀

Lemma 8 follows from Lemmas 13 and 15.

Proof of Lemma 8. We fix the execution up to the end of round t, so we do not have to condition on $\mathcal{F}_t$. We have that

$$\begin{aligned}
&\mathbf{Pr}\left[\tau^{(v)}(t) > k\right] \\
&= \mathbf{Pr}\left[\tau^{(v)}(t) > k \wedge G_{t,k}(v) > 0.05k\right] + \mathbf{Pr}\left[\tau^{(v)}(t) > k \wedge G_{t,k}(v) \leq 0.05k\right] \\
&\leq \mathbf{Pr}\left[\tau^{(v)}(t) > k \wedge G_{t,k}(v) > 0.05k\right] + e^{-k/100} \qquad (6)\\
&\leq (1 - e^{-27})^{0.05k} + e^{-k/100} \qquad (7)\\
&\leq e^{-e^{-27}0.05k} + e^{-k/100} \leq e^{-e^{-29}k} + e^{-k/100} \leq e^{-e^{-30}k},
\end{aligned}$$

where (6) follows from Lemma 13, and (7) follows from Lemma 15. ◀

6.2 Stopping Times for Platinum Rounds (Proof of Lemma 9)

Proof of Lemma 9. We fix the execution up to the end of round t, so we do not have to condition probabilities on $\mathcal{F}_t$.

We prove part (a) first. Since $u \in PM_t \setminus S_t$, we have $\ell_t(u) \leq 0$. Since $t > \max_{w \in V} \ell_{\max}(w)$ Lemma 5 implies that $\mu_t(u) > 0$, i.e., $\ell_t(w) > 0$ for all $w \in N(v)$. By Lemma 7 there exist a round $t' \leq t$ with $\ell_{t'}(u) = -\ell_{\max}(u)$ and $t - \ell_{\max}(u) \leq t'$. Thus, each neighbor w of u incremented its level during the rounds $t', t'+1, \ldots, t$ or the level of w reached $\ell_{\max}(u)$. Let $\ell = t - t'$. Thus, $\ell_t(w) \geq \min\{\ell_{\max}(w), \ell_{\max}(u) - \ell\}$. Hence, if $\ell_t(w) < \ell_{\max}(w)$ then $p_t(w) \leq 2^{-(\ell_{\max}(u) - \ell)}$. This yields

$$d_t(u) = \sum_{w \in N(u) \setminus S_t} p_t(w) \leq \sum_{w \in N(u) \setminus S_t} 2^{-(\ell_{\max}(u) - \ell)},$$

and also that, in the subsequent ℓ rounds, vertex u is beeping and the level of each of u's neighbors increases in each round. Therefore, we have $\ell_{t+\ell}(w) \geq \min\{\ell_{\max}(w), \ell_{\max}(u)\}$ for each $w \in N(u)$, and, moreover

$$d_{t+\ell}(u) = \sum_{\substack{w \in N(u) \setminus S_t: \\ \ell_{\max}(w) > \ell_{\max}(u)}} 2^{-\ell_{\max}(u)} = \eta'_t(u).$$

We notice that, if $\ell_{\max}(u) \geq \ell_{\max}(w)$ for each $w \in N(u)$, then we have that $d_{t+\ell}(u) = 0$ and hence $\mathbf{Pr}\left[\sigma^{(u)}(t) = \sigma_{\text{in}}^{(u)}(t) \wedge \sigma^{(u)}(t) \leq \ell_{\max}(u)\right] = 1$, and this proves (a) when $\eta'_t(u) = 0$. If otherwise $\eta'_t(u) > 0$, we can define

$$\ell' = \max_{w \in N(u)} \ell_{\max}(w) - \ell_{\max}(u)$$

which is such that $0 < \ell' \leq \max_{w \in N(u)} \ell_{\max}(w)$, and we have that $(\sigma^{(u)}(t) = \sigma^{(u)}_{\text{in}}(t)) \wedge (\sigma^{(u)}(t) \leq \ell')$ happens with probability at least

$$\begin{aligned}\prod_{i=1}^{\ell'} \prod_{\substack{w \in N(u): \\ \ell_{\max}(w) > \ell_{\max}(u)}} \left(1 - \frac{p_t(w)}{2^{i+\ell_{\max}(u)}}\right) &\geq \prod_{i=1}^{\ell'} \prod_{\substack{w \in N(u): \\ \ell_{\max}(w) > \ell_{\max}(u)}} 3^{-p_t(w)2^{-(i+\ell_{\max}(u))}} \\ &\geq \prod_{i=1}^{\ell'} 3^{-\eta'_t(u)2^{-i}} \\ &\geq 3^{-\eta'_t(u)},\end{aligned}$$

where the first inequality follows from the fact that, for each $w \in N(u)$, $p_t(w)/2^{\ell_{\max}(u)} \leq 2^{-4}$.

Next we prove part (b). We observe that, for each $x \geq 0$, we have that

$$\begin{aligned}&\mathbf{Pr}\left[\sigma^{(u)}(t) \neq \sigma^{(u)}_{\text{in}}(t) \wedge \sigma^{(u)}(t) > \ell_{\max}(u) + x\right] \\ &\leq \mathbf{Pr}\left[\sigma^{(u)}(t) = \sigma^{(u)}_{\text{out}}(t) \mid \sigma^{(u)}(t) > \ell_{\max}(u) + x\right].\end{aligned}$$

Since we have that the event $\sigma^{(u)}(t) > \ell_{\max}(u) + x$ implies that, for each $w \in N(u)$, $p_{t+\ell_{\max}(u)+x}(w) \leq 2^{-(\ell_{\max}(u)+x)}$, we have that

$$\begin{aligned}\mathbf{Pr}\left[\sigma^{(u)}(t) = \sigma^{(u)}_{\text{out}}(t) \mid \sigma^{(u)}(t) > \ell_{\max}(u) + x\right] &\leq 1 - \prod_{w \in N(u) \setminus S_t} \left(1 - 2^{-(\ell_{\max}(u)+x)}\right) \\ &\leq \sum_{w \in N(u) \setminus S_t} 2^{-(\ell_{\max}(u)+x)} \\ &\leq \eta'_t(u) 2^{-x}.\end{aligned}$$ ◀

7 Conclusion

In this paper, we describe a new randomized self-stabilizing MIS algorithm using the beeping model requiring each vertex to have only limited knowledge about the topology that comes close to the $O(\log n)$ time bound. The algorithm is motivated by the non self-stabilizing algorithm of Jeavons et al. [18]. To transform it into a self-stabilizing algorithm we had to overcome two issues: Firstly, the known initial configuration and secondly, the phase concept. We prove that the global knowledge of the maximum degree is sufficient to obtain a $O(\log n)$ self-stabilizing algorithm. If we rely on the local knowledge of the vertex degree, the algorithm stabilizes in time $O(\log n \cdot \log \log n)$. It is an open question if this upper bound is tight, or whether the analysis can be improved to obtain the upper bound $O(\log n)$.

We remark that, for a beeping model with two channels, we can easily implement the phases with two rounds with the presence of two beeping channels, and we prove that, in such a case, a self-stabilizing MIS algorithm terminating in $O(\log n)$ relies on the (almost) local knowledge of the 2-hop neighbors. It is natural to ask whether the local knowledge can be completely removed, obtaining an algorithm for the beeping model (with one or two channels) that computes an MIS in a self-stabilizing way.

References

1 Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. *Distributed Comput.*, 26(4):195–208, 2013. doi:10.1007/s00446-012-0175-7.

2 Yehuda Afek, Noga Alon, Omer Barad, Eran Hornstein, Naama Barkai, and Ziv Bar-Joseph. A biological solution to a fundamental distributed computing problem. *Science*, 331(6014):183–185, 2011. doi:10.1126/science.1193210.

3 Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 7(4):567–583, 1986. doi:10.1016/0196-6774(86)90019-2.

4 Alkida Balliu, Sebastian Brandt, Juho Hirvonen, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. Lower bounds for maximal matchings and maximal independent sets. *J. ACM*, 68(5):39:1–39:30, 2021. doi:10.1145/3461458.

5 Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In *Proc. 24th International Symposium on Distributed Computing, DISC*, pages 148–162, 2010. doi:10.1007/978-3-642-15763-9_15.

6 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974. doi:10.1145/361179.361202.

7 Shlomi Dolev. *Self-Stabilization.* MIT Press, 2000. URL: http://www.cs.bgu.ac.il/%7Edolev/book/book.html.

8 Yuval Emek and Eyal Keren. A thin self-stabilizing asynchronous unison algorithm with applications to fault tolerant biological networks. In *Proc. 40th ACM Symposium on Principles of Distributed Computing, PODC*, pages 93–102. ACM, 2021. doi:10.1145/3465084.3467922.

9 Yuval Emek and Jara Uitto. Dynamic networks of finite state machines. *Theor. Comput. Sci.*, 810:58–71, 2020. doi:10.1016/J.TCS.2017.05.025.

10 Yuval Emek and Roger Wattenhofer. Stone age distributed computing. In *Proc. 32nd ACM Symposium on Principles of Distributed Computing, PODC*, pages 137–146, 2013. doi:10.1145/2484239.2484244.

11 Salwa Faour, Mohsen Ghaffari, Christoph Grunau, Fabian Kuhn, and Václav Rozhon. Local distributed rounding: Generalized to MIS, matching, set cover, and beyond. In *Proc. 34th ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 4409–4447, 2023. doi:10.1137/1.9781611977554.CH168.

12 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proc. 27th ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 270–277, 2016. doi:10.1137/1.9781611974331.ch20.

13 Mohsen Ghaffari. Distributed MIS via all-to-all communication. In *Proc. 36th ACM Symposium on Principles of Distributed Computing, PODC*, pages 141–149, 2017. doi:10.1145/3087801.3087830.

14 Mohsen Ghaffari. Distributed maximal independent set using small messages. In *Proc. 30th ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 805–820, 2019. doi:10.1137/1.9781611975482.50.

15 Mohsen Ghaffari, Christoph Grunau, and Václav Rozhon. Improved deterministic network decomposition. In *Proc. 32nd ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 2904–2923, 2021. doi:10.1137/1.9781611976465.173.

16 George Giakkoupis, Volker Turau, and Isabella Ziccardi. Luby's MIS algorithms made self-stabilizing. *Information Processing Letters*, 188:106531, 2025. doi:10.1016/j.ipl.2024.106531.

17 George Giakkoupis and Isabella Ziccardi. Distributed self-stabilizing MIS with few states and weak communication. In *Proc. 42nd ACM Symposium on Principles of Distributed Computing, PODC*, pages 310–320, 2023. doi:10.1145/3583668.3594581.

18 Peter Jeavons, Alex Scott, and Lei Xu. Feedback from nature: Simple randomised distributed algorithms for maximal independent set selection and greedy colouring. *Distributed Comput.*, 29(5):377–393, 2016. doi:10.1007/S00446-016-0269-8.

19 Nathan Linial. Distributive graph algorithms - Global solutions from local data. In *Proc. 28th Symposium on Foundations of Computer Science, FOCS*, pages 331–335, 1987. doi:10.1109/SFCS.1987.20.

20 Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. MST construction in O(log log n) communication rounds. In *Proc. 15th Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 94–100, 2003. doi:10.1145/777412.777428.

21 Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986. doi:10.1137/0215074.

22 Thomas Moscibroda and Roger Wattenhofer. Maximal independent sets in radio networks. In *Proc. 24th ACM Symposium on Principles of Distributed Computing, PODC*, pages 148–157, 2005. doi:10.1145/1073814.1073842.

23 David Peleg. *Distributed computing: A locality-sensitive approach.* SIAM, 2000.

APPENDIX

A Tools

▶ **Lemma 16.** *Let $X_1, \dots, X_n$ be a sequence of i.i.d. geometric random variables with success probability p. Then, we have that*

$$\mathbf{Pr}\left[\sum_{i=1}^{n} X_i \geq k\right] = \mathbf{Pr}\left[Bin(k,p) \leq n\right].$$

Proof. Asking that $\sum_{i=1}^{n} X_i \geq k$ is like asking that, in k Bernoulli trials, we have less than n successes. ◀

▶ **Theorem 17** (Chernoff's Inequality). *Let $X = \sum_{i=1}^{n} X_i$, where X_i with $i \in [n]$ are independently distributed in $[0,1]$. Let $\mu = \mathbf{E}[X]$ and $\mu_- \leq \mu \leq \mu_+$. Then:*

(a) *for every $t > 0$*

$$\mathbf{Pr}[X > \mu_+ + t] \leq e^{-2t^2/n} \quad \textit{and} \quad \mathbf{Pr}[X < \mu_- - t] \leq e^{-2t^2/n};$$

(b) *for $0 < \epsilon < 1$*

$$\mathbf{Pr}[X > (1+\epsilon)\mu_+] \leq e^{-\frac{\epsilon^2}{3}\mu_+} \quad \textit{and} \quad \mathbf{Pr}[X < (1-\epsilon)\mu_-] \leq e^{-\frac{\epsilon^2}{2}\mu_-}.$$

B Two Beeping Channels (Proof of Corollary 4)

One of the reasons that the MIS algorithm of Jeavons et al. [18] is not self-stabilizing is the usage of phases consisting of two rounds. This allows a newly joined MIS vertex to signal this event to all neighbors in the subsequent round. Afterwards, these vertices become passive, i.e., no longer participate in the algorithm. Thus, a vertex v that newly joined the MIS cannot be forced by a neighbor that is unaware that v joined the MIS to leave the MIS again in the successive round. This problem can be circumvented if a second beeping channel is available, since we can let the vertices joining the MIS beep on that channel. Indeed, beginning in the round immediately following the round vertex v joined the MIS, it signals in every round on this second channel. Neighbors of v take this as an opportunity to become non-MIS vertices. This second channel and the corresponding behavior can be easily integrated into Algorithm 1 (see Algorithm 2). The range of state variable $\ell(v)$ is restricted to $[0, \ell_{\max}(v)]$, where $\ell(v) = 0$ (resp. $\ell(v) = \ell_{\max}(v)$) implies that v is in the MIS (resp. not in the MIS). If a vertex v which is enabled to signal with $beep_1$ receives neither signal from a neighbor then it sets $\ell(v)$ to 0 and signals $beep_2$ in all future rounds. Vertices receiving a $beep_2$ signal set $\ell(v)$ to $\ell_{\max}(v)$ and refrain from beeping in future rounds. We end this section by giving the proof of Corollary 4.

Algorithm 2 Self-stabilizing version of Jeavons at al.'s algorithm with two beeping channels.

```
state: $\ell \in \{0, \dots, \ell_{\max}(v)\}$
in each round $t = 1, 2, \dots$ do
    if $0 < \ell < \ell_{\max}(v)$ then
        $beep_1 \leftarrow true$ with probability $2^{-\ell}$ and $beep_1 \leftarrow false$ otherwise
    else $beep_1 \leftarrow false$
    $beep_2 \leftarrow (\ell = 0)$
    if $beep_1$ or $beep_2$ then send the corresponding signal to all neighbors
    receive any signals sent by neighbors

    if $beep_2$ signal received then
        $\ell \leftarrow \ell_{\max}(v)$;
    else if $beep_1$ signal received then
        $\ell \leftarrow \min\{\ell + 1, \ell_{\max}(v)\}$
    else if $beep_1$ then
        $\ell \leftarrow 0$
    else if $beep_2 = false$ then
        $\ell \leftarrow \max\{\ell - 1, 1\}$
```

Proof of Corollary 4. We consider Algorithm 2 and we notice that the update rule of ℓ of the non-stable vertices is the same of Algorithm 1, and hence we can still use Lemma 8, since it relies just on the update rule for ℓ. Note the difference between the two algorithms: In Algorithm 1 if the level of a vertex is 0 or lower then it is guaranteed that it sends a beep. In Algorithm 2 a vertex sends a $beep_2$ signal if and only if its level is 0.

We will prove that the termination time of Algorithm 2 is $O(\log n)$, if we take $\ell_{\max}(v) \geq 2 \log \deg_2(v) + 15$ for every $v \in V$. We first notice that, in this case, we have that

$$\eta_1(v) \leq \sum_{u \in N(v)} 2^{-2 \log \deg_2(u) - 15} \leq \sum_{u \in N(v)} \frac{1}{\deg^2(v)} 2^{-15} \leq 0.0001,$$

and hence, for each $t \geq 1$ and $v \in V$ we have that $\eta_t(v) \leq 0.0001$.

We notice that, for a vertex $v \in V$ to stabilize in Algorithm 2, it suffice to have a platinum round for v. Hence, from Lemma 8 we have that each vertex stabilizes in time $O(\log n)$ with probability at least $1 - 1/n^2$. The theorem follows from the union bound applied to all vertices. ◀

C Omitted Proofs

Proof of Lemma 5. Let t_0 be the first round such that $\ell_{t_0}(v) > 0$ or $\mu_{t_0}(v) > 0$. First, we will prove that this condition continues to hold for all rounds $t \geq t_0$. Then, we will prove that $t_0 \leq \max_{w \in V} \ell_{\max}(w) + 1$.

Consider any round $t \geq t_0$ and assume that $\ell_t(v) > 0$ or $\mu_t(v) > 0$. This implies that $\mu_{t+1}(v) > 0$ or $\ell_{t+1}(v) > 0$. Indeed, assume that $\mu_t(v) \leq 0$. Then $\ell_t(v) > 0$ and at least one neighbor of v beeps in round t. Thus, $\ell_{t+1}(v) = \min\{\ell_t(v) + 1, \ell_{\max}(v)\} \geq \ell_t(v) > 0$, i.e., the condition of the lemma holds in round $t + 1$. Next consider the case that $\mu_t(v) > 0$. If v beeps in round t then all neighbors increase their value for ℓ, i.e., $\mu_{t+1}(v) \geq \mu_t(v) > 0$.

If v does not beep in round t then $\ell_t(v) > 0$. Indeed, if no neighbor of v beeps then $\ell_{t+1}(v) = \max\{\ell_t(v) - 1, 1\} > 0$, and if at least one neighbor of v beeps then $\ell_{t+1}(v) = \min\{\ell_t(v) + 1, \ell_{\max}(v)\} \geq \ell_t(v) > 0$, i.e., the condition of the lemma holds in round $t + 1$.

Assume that $\ell_0(v) \leq 0$ and $\mu_0(v) \leq 0$. Then, in the first round all vertices in $N^+(v)$ beep. Hence, all these vertices increment their level by 1, i.e., $\ell_1(v) = \ell_0(v) + 1$ and $\mu_1(v) = \min_{u \in N(v)} \frac{\ell_0(u)+1}{\ell_{\max}(u)}$. Since $-\ell_{\max}(u) \leq \ell_0(u)$ for all vertices $u \in V$, there exists $t_0 \leq \max_{u \in N^+(v)} \ell_{\max}(u) + 1$, such that $\ell_{t_0}(v) > 0$ or $\mu_{t_0}(v) > 0$. This completes the proof. ◀

Proof of Lemma 13. Fix a vertex $v \in V$. We consider $k \geq 70\ell_{\max}(v)$ consecutive rounds, starting from a round t which is not a platinum round of v. Since $\eta_t(v)$ is decreasing in t, in all rounds $t + m$, $m \geq 0$, we have $\eta_{t+m}(v) < 0.0001$. We consider the following sets of rounds

$$\begin{aligned}
D_{t,k}(v) &= \{0 \leq m \leq k : d_{t+m}(v) > 0.2\}\\
E_{t,k}(v) &= \{0 \leq m \leq k : d_{t+m}(v) > 0.1 \text{ and } d^L_{t+m}(v) \geq 0.1 d_{t+m}(v)\}\\
F_{t,k}(v) &= \{0 \leq m \leq k : d_{t+m}(v) > 0.1 \text{ and } d^L_{t+m}(v) < 0.1 d_{t+m}(v)\}\\
H_{t,k}(v) &= \{0 \leq m \leq k : d_{t+m}(v) < 0.2 \text{ and } \ell_{t+m}(v) \leq 1\}.
\end{aligned}$$

We say that in some round t' we have a *wrong move* if none of the following conditions occurs

(a) t or $t + 1$ is a platinum round of v;
(b) $\eta_t(v) > 0.0001$;
(c) $d_t(v) \leq 0.01$ or $d_t(v) > 0.02$;
(d) $d^L_t(v) \geq 0.01 d_t(v)$;
(e) $d_{t+1}(v) < 0.6 d_t(v)$;
(f) $\ell_{t+1}(v) \leq \max\{\ell_t(v) - 1, 1\}$;

From Lemma 14, we have that a vertex makes a wrong move with probability at most 0.03. Since the randomness of each round is independent of the others, we know by Chernoff's bound, that in the rounds $\{t, t + 1, \ldots, t + k\}$ there are at most $0.04k$ wrong moves with probability at least $1 - e^{-k/100}$, and we will refer to this event with B.

In the rest of the proof, we assume that B happens, and we will see that it implies, deterministically, that $\tau^{(v)}(t) \leq k$ or that $G_{t,k}(v) \geq 0.1k$. So, we assume that $\tau^{(v)}(t) > k$ and we will prove that, under event B, this implies that $G_{t,k}(v) \geq 0.1k$. We remark that, if $\tau^{(v)}(t) > k$, for each $0 \leq m \leq k$, we have that $d_{t+m+1}(v) \leq 2d_{t+m}(v)$.

In what follows, we will prove that:

(i) if $E_{t,k}(v) < 0.05k$, then $D_{t,k}(v) < 0.25k$
(ii) if $D_{t,k}(v) < 0.25k$, then $H_{t,k}(v) > 0.28k$.

We prove (i) first. We denote with $D'_{t,k}(v)$ the set $\{0 \leq m \leq k : d_{t+m}(v) > 0.1\}$ and we notice that $D'_{t,k}(v) = E_{t,k}(v) \cup F_{t,k}(v)$. Also let $h = |D_{t,k}(v)|$ and $h' = |D'_{t,k}(v)|$. Since the number of wrong moves is bounded by $0.04k$, and since $E_{t,k}(v) < 0.05k$ the number of rounds in $D'_{t,k}(v)$ in which $d_{t+m}(v)$ can double is at most $0.09k$, and in the rest of the rounds it will decrease of a factor of 0.6.

In order to keep $d_{t+m}(v) > 0.2$, in a consecutive interval of rounds in $D'_{t,k}(v)$, the number of increasing moves must be at least $\log_{0.5}(0.6) > 0.7$ times the number of decreasing moves, and at most $\log_{5/3}(10d_t(v)) \leq 2\log(10\deg(v)) \leq 2\log\deg(v) + 8$ decreases are used to decrease the initial value of $d_t(v)$ below 0.1. Hence, the total number of rounds in $D_{t,k}(v)$ is at most

$$\begin{aligned}0.09k + \frac{0.09}{0.7}k + 2\log(\deg(v)) + 8 &\leq 0.22k + 2\log\deg(v) + 8\\ &\leq 0.22k + 2\ell_{\max}(v)\\ &\leq 0.22k + 0.03k\\ &= 0.25k.\end{aligned}$$

Next we prove (ii). Since $|D_{t,k}(v)| \leq 0.25k$, the set $D^C_{t,k}(v) = \{0 \leq m \leq k : d_{t+m}(v) \leq 0.2\}$ contains at least $0.75k$ rounds. The number of wrong moves is bounded by $0.04k$, and in rounds $D^C_{t,k}(v)$ a wrong moves implies that $\ell_{t+m+1}(v) = \min\{\ell_{t+m}(v)+1, \ell_{\max}(v)\}$. Moreover, we have that in the rounds $D_{t,k}(v)$, $\ell_{t+m+1}(v) \leq \min\{\ell_{t+m}(v)+1, \ell_{\max}(v)\}$ is satisfied. Hence, $\ell_t(v)$ can increase in at most $|D_{t,k}(v)| + 0.04k \leq 0.29k$ rounds. The rounds in $D^C_{t,k}(v)$ in which no wrong move occurred are such that $\ell_{t+m+1}(v) = \max\{1, \ell_{t+m}(v) - 1\}$, since we assumed that $\tau^{(v)}(t) > k$. Since $D^C_{t,k}(v)$ has at least $0.75k$ elements, and since the number of wrong moves is bounded by $0.04k$, the number of moves in which $\ell_{t+m}(v)$ decreases is at least $0.75k - 0.04k = 0.71k$. Since the number of rounds in which $\ell_{t+m}(v)$ increases is at most $0.29k$, we have that the number of increases is at least 2.4 times the number of decreases.

Denote the number of rounds in $D^C_{t,k}(v)$ where $\ell_{t+m}(v)$ decreases by U and those where it increases by D. Thus, $D + U \geq 0.75k$ and $U \geq 2.4D$. In the worst case, each round with an increase follows a round with a decrease. Then, we still have $0.75k-2D$ rounds with an increase left. Then, $0.75k - 2D = U - D \geq 0.58U \geq 0.3k$. As it takes at most $\ell_{\max}(v)$ for $p_{t+m}(v)$ to reach $1/2$ we can say that, since $k \geq 70\ell_{\max}(v)$, we have at least $0.3k - \ell_{\max}(v) > 0.28k$ rounds where $\ell_{t+m}(v) = 1$ and $d_{t+m}(v) < 0.2$, hence $H_{t,k}(v) > 0.28k$. ◀

D Illustration of Beeping Probability

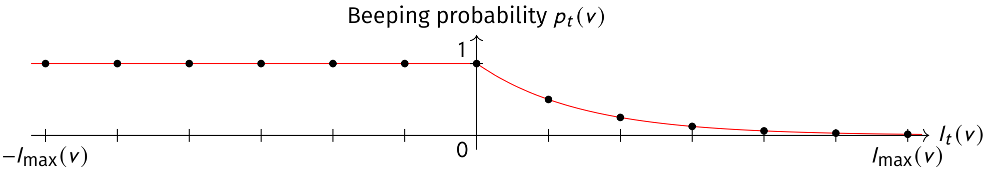


Figure 1 Beeping probability $p_t(v)$ of v based on value of $\ell_t(v)$.

Massively Parallel Ruling Set Made Deterministic

Jeff Giliberti
University of Maryland, College Park, MD, USA

Zahra Parsaeian
University of Freiburg, Germany

Abstract

We study the *deterministic complexity* of the 2-Ruling Set problem in the model of Massively Parallel Computation (MPC) with linear and strongly sublinear local memory.

Linear MPC: We present a constant-round deterministic algorithm for the 2-Ruling Set problem that matches the randomized round complexity recently settled by Cambus, Kuhn, Pai, and Uitto [DISC'23], and improves upon the deterministic $O(\log \log n)$-round algorithm by Pai and Pemmaraju [PODC'22]. Our main ingredient is a simpler analysis of CKPU's algorithm based solely on bounded independence, which makes its efficient derandomization possible.

Sublinear MPC: We present a deterministic algorithm that computes a 2-Ruling Set in $\tilde{O}(\sqrt{\log n})$ rounds deterministically. Notably, this is the first deterministic ruling set algorithm with sublogarithmic round complexity, improving on the $O(\log \Delta + \log \log^* n)$-round complexity that stems from the deterministic MIS algorithm of Czumaj, Davies, and Parter [TALG'21]. Our result is based on a simple and fast randomness-efficient construction that achieves the same sparsification as that of the randomized $\tilde{O}(\sqrt{\log n})$-round LOCAL algorithm by Kothapalli and Pemmaraju [FSTTCS'12].

2012 ACM Subject Classification Theory of computation → MapReduce algorithms

Keywords and phrases deterministic algorithms, distributed computing, massively parallel computation, graph algorithms, derandomization

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.29

Related Version *Full Version*: `https://arxiv.org/abs/2406.12727`

Funding *Jeff Giliberti*: Financial support in part by the Fulbright U.S. Graduate Student Program, sponsored by the U.S. Department of State and the Italian-American Fulbright Commission. The content does not necessarily represent the views of the Program.

Acknowledgements We are grateful to Christoph Grunau and Manuela Fischer for valuable discussions. We would also like to thank the anonymous reviewers for their helpful feedback, and Yannic Maus for his shepherding of the paper.

1 Introduction

In this paper, we present faster deterministic parallel algorithms for finding 2-ruling sets. Given an n-vertex m-edge graph $G = (V, E)$ and an integer $\beta \geq 1$, the more general problem of β-ruling sets consists of finding a subset $S \subseteq V$ of non-adjacent vertices such that each vertex $v \in V \setminus S$ is at most β hops away from some vertex in S. Thus, a β-ruling set is also a $\beta + 1$ ruling set. This concept serves as a natural generalization of one of the most central and well-studied problems in distributed graph algorithms, known as *Maximal Independent Set* (MIS), which corresponds to a 1-ruling set. Generally, for $\beta \geq 1$, the complexity of a β-ruling set reduces as the value of β increases.

We design 2-ruling set algorithms for the model of Massively Parallel Computation (MPC) in the strongly sublinear and linear memory regimes. The study of 2-ruling sets is motivated by its close relationship with MIS, while still permitting the development of considerably faster algorithms. Additionally, it is known that for problems utilizing MIS as a subroutine, a β-ruling set may serve as an alternative for some $\beta > 1$ [4].

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 29; pp. 29:1–29:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

MPC Model. Initially introduced by [32] and later refined in [2, 7, 30], this model is characterized by a set of M machines each with memory S. The input is distributed across machines and the computation proceeds in synchronous rounds. Each round machines perform arbitrary local computation and all-to-all communication, sending and receiving up to S words. The main goal is to minimize the number of communication rounds required by the algorithm. A second goal is to minimize the global space needed to solve the problem, i.e., the number of machines times the local memory per machine, which is $\Omega(n+m)$ for graph problems. In the linear regime of MPC each machine is assigned local memory $S = O(n)$, while in the (strongly) sublinear regime of MPC the local memory is $O(n^\alpha)$, for constant $0 < \alpha < 1$.

Linear MPC. In the linear model of MPC, a series of works showed that several fundamental problems such as $(\Delta + 1)$-coloring [13, 16] and minimum-spanning tree [42] admit constant-round deterministic algorithms. Surprisingly, a recent work of [11] provides a randomized 2-ruling set algorithm with constant-round complexity improving on the $O(\log\log\log n)$ time algorithm by [31] and the $O(\log\log\Delta)$ time bound that stems from the MIS algorithm by [26]. On the deterministic side, [43] gave an algorithm that computes a 2-ruling set in $O(\log\log n)$ time, which improved on the $O(\log\Delta + \log\log^* n)$ round complexity due to the deterministic MIS algorithm of [18, 17, 20]. Key challenges in this domain lie in determining the existence of deterministic algorithms achieving constant-round complexity for 2-ruling sets and sublogarithmic-round complexity for MIS.

Sublinear MPC. In the sublinear model of MPC, the above $O(\log\Delta + \log\log^* n)$-round algorithm by [18, 17] is the fastest known for both MIS and 2-ruling set. On the randomized side, [28] show that MIS can be solved in $\tilde{O}(\sqrt{\log\Delta} + \log\log n)$ rounds and [43] show that 2-ruling set can be solved in $\tilde{O}(\log^{1/6}\Delta + \log\log n)$, where the $\tilde{O}(\cdot)$ notation hides $\operatorname{poly}\log(\cdot)$ factors. It may be worth noting that if we limit the global space to $\tilde{O}(n+m)$, then the fastest 2-ruling set algorithm has $\tilde{O}(\log^{1/4} n + \log\log n)$ randomized complexity [43] and $O(\log\Delta\log\log n)$ deterministic complexity [18, 23].

Other Related Work. There is a large body of work studying ruling sets in the LOCAL model [24, 9, 31, 45, 10, 6]. The most relevant to ours is the randomized LOCAL algorithm of [35] for computing 2-ruling sets that combined with [25] yields a LOCAL round complexity of $\tilde{O}(\sqrt{\log n})$. On the hardness side, in the LOCAL model, there is a lower bound for 2-ruling set of $\Omega(\min\{\sqrt{\Delta}, \log_\Delta n\})$ deterministic rounds and of $\Omega(\min\{\sqrt{\Delta}, \log_\Delta \log n)$ randomized rounds by [5, 4], which, in terms of its proportion to n, are $\Omega(\frac{\log n}{\log\log n})$, and $\Omega(\frac{\log\log n}{\log\log\log n})$, respectively. For MIS and maximal matching (MM), the best known deterministic lower bound is $\Omega(\min\{\Delta, \log_\Delta n\})$ by [3], and the best known randomized lower bounds are $\Omega(\min\{\Delta, \log_\Delta \log n\})$ by [3] and $\Omega(\min\{\frac{\log\Delta}{\log\log\Delta}, \log_\Delta n\})$ by [36], which, in terms of its proportion to n, are $\Omega(\frac{\log n}{\log\log n})$, $\Omega(\frac{\log\log n}{\log\log\log n})$, and $\Omega(\sqrt{\frac{\log n}{\log\log n}})$, respectively. Via the MPC conditional lower-bound framework by [27, 17], these results give the following component-stable lower bounds for sublinear MPC algorithms:

- $\Omega(\log\log n)$ for deterministic 2-ruling set, deterministic and randomized MIS and MM.
- $\Omega(\log\log\log n)$ for randomized 2-ruling set.

1.1 Our Contribution

We design improved deterministic algorithms for the problem of 2-ruling set in the MPC setting with linear and sublinear local memory.

Linear MPC Regime. We develop a deterministic algorithm that matches the constant-round complexity of [11] and even its optimal global space usage.

▶ **Theorem 1.** *There is a $O(1)$-round linear MPC algorithm that computes a 2-ruling set deterministically using linear global space.*

Prior to our work, the best known deterministic complexity was $O(\log\log n)$ by a result of [43]. Our algorithm (Section 3) is obtained by derandomizing the $O(1)$-round algorithm of [11]. While the derandomization framework of our algorithm has been applied successfully to numerous MPC graph problems [12, 16, 19, 18, 15, 22, 23, 43], the main challenge lies in analyzing (a slight variation of) [11]'s algorithm under limited independence, as we overview later in Section 1.2.1.

Sublinear MPC Regime. We design the first deterministic sublogarithmic algorithm for finding a 2-ruling set when the memory per machine is strictly sublinear.

▶ **Theorem 2.** *There is a deterministic sublinear MPC algorithm that finds a 2-ruling set in $O(\sqrt{\log\Delta}\cdot\log\log\Delta+\log\log^* n)$ rounds using $O(n^{1+\varepsilon}+m)$ global space, for any constant $\varepsilon>0$. Moreover, the same algorithm runs in $O(\sqrt{\log\Delta}\cdot\log\log n)$ using global space $O(n+m)$.*

For $\Delta\gg\log^* n$, our algorithm gives an almost quadratic improvement over the runtime obtained using the MIS algorithm of [20], and gets closer to the $\tilde{O}(\log^{1/6}\Delta+\log\log n)$ randomized complexity of [33]. It is worth noting that it matches the conditionally-optimal runtime of $\Omega(\log\log n)$ when $\Delta=O(2^{\log^2\log n/\log\log\log n})$, even though, being it not component-stable, the lower bound does not apply.

This algorithm (Section 4) is obtained by derandomizing the sparsification developed by [35] for solving 2-ruling sets in the LOCAL model. Specifically, we show that a randomized $O(1)$-LOCAL downsampling step can be carried out in only $O(\log\log\Delta)$ rounds deterministically in MPC with strongly sublinear space per machine and optimal global space. To achieve that, we combine several well-established derandomization tools such as limited independence, the method of conditional expectation, and coloring for reducing seed length, as we discuss in Section 1.2.2.

We also note that our techniques may be more general and apply to β-ruling sets for $\beta>2$. Concretely, one may combine our result with the framework of [10] to obtain faster MPC β-ruling sets algorithms. This direction is left for future work.

1.2 2-Ruling Sets: Technical Overview

We present the main intuition behind the recent constant-round randomized algorithm by [11] in the linear regime of MPC and the randomized $\tilde{O}(\sqrt{\log n})$-round LOCAL algorithm by [35], which is also closely followed by subsequent works [31, 33, 43]. Then, we provide an overview of our deterministic algorithms and the main ideas that lead to randomness-efficient analyses.

1.2.1 Linear Memory Regime

Randomized Constant-Round Algorithm. The constant-round 2-ruling set algorithm by [11] relies on computing an MIS iteratively on subgraphs of linear size, which can be solved locally on a single machine. Their algorithm samples each vertex v from V and includes it in V_{samp} independently with probability $1/\sqrt{\deg(v)}$. This sampling primitive is shown to give two useful structural properties, with high probability. First, the induced subgraph $G[V_{\text{samp}}]$ has a linear number of edges. Second, a certain MIS computation on $G[V_{\text{samp}}]$ returns an independent set that is at distance at most two from all but at most $n/\sqrt{d}$ vertices with degree $[d, 2d)$ in the original graph G, for each $d \in \{2^{\lfloor \log \Delta \rfloor}, 2^{\lfloor \log \Delta \rfloor - 1}, \ldots, \Omega(1)\}$. Then, it is shown that, after two repetitions, the number of remaining edges for each degree class d is at most $n/\text{poly}(d)$, which sums up to $O(n)$ over all d's.

Their analysis of the above sampling process relies on full independence in the sense that random decisions of any node influence its neighbors at distance at most three. Then, each node influences only up to $n^{3\alpha}$ many nodes by assuming that any node has degree at most n^{α}, for constant $\alpha > 0$. This property is exploited to union bound over large sets of independent nodes in G^7, since nodes at distance 8 are enough far apart not to influence one another. Clearly, this property breaks apart under our constraint of limited independence and requires to analyze the sampling process differently.

Constant-Round Derandomization. In a nutshell, we show that the same asymptotic guarantees as that provided by the above randomized algorithm can be achieved deterministically. While it is easy to show that their initial sampling step gives a subgraph with a linear number of edges in expectation, even under pairwise independence, the main challenge is to prove that only $n/d^{\Omega(1)}$ nodes survive across all $O(\log \Delta)$ d-degree classes, simultaneously. Establishing the same polynomial decrease (in $d^{\Omega(1)}$) of the size of each d-degree class ensures the same constant-round complexity.

Our key modification to [11]'s analysis is to increase the threshold for a node to be called good. We say that a node of degree d is good if it has at least $d^{\Omega(1)}$ neighbors in $G[V_{\text{samp}}]$, as opposed to the $\Theta(\log n)$ requirement of [11]. This leads to the following two properties.

First, in the sampling step, we prove that each good node of degree d is covered with probability $1 - 1/\text{poly}(d)$ and that suffices. In fact, through the method of conditional expectation, non-covered nodes will induce at most $O(n)$ edges.

Second, in the MIS step, we prove that remaining "bad" nodes are at most $n/d^{\Omega(1)}$ for each degree class, without any assumption on the maximum degree. To achieve that, we combine a pairwise independent MIS algorithm (similar to that of [23]) with a pessimistic estimator that notably expresses the progress made over *all* degree classes as a *single* expectation. This expectation can then be obtained by means of standard derandomization tools.

1.2.2 Strongly Sublinear Memory Regime

Randomized 2-Ruling Set Sparsification. The central step of the 2-ruling set algorithms by [34, 33] is a sparsification procedure that returns a subgraph G' of sufficiently small maximum degree. Then, computing a maximal independent set on G' has time proportional to its maximum degree, and yields a 2-ruling set that covers all vertices in G which have a neighbor in G'.

They construct a subgraph G' of maximum degree $O(f \cdot \log n)$ such that any (high-degree) node with a degree in $[\Delta, \Delta/f]$ in G has a neighbor in G', for some parameter $f \geq \log n$. It is easy to see that sampling each vertex $v \in V$ with probability $f \cdot \log n/\Delta$ independently ensures that every vertex with degree at least Δ/f will have a sampled vertex in its neighborhood with high probability.

We just focused solely on covering vertices with degrees in $[\Delta, \Delta/f]$. It turns out that, by each time removing the subgraph G' and its neighbors, the same sampling step can be repeated $O(\log_f \Delta)$ times, where in the j-th step nodes with degrees in $[f^{\log_f \Delta-(j-1)}, f^{\log_f \Delta-j}]$ are covered, with $j \in [\log_f \Delta]$. This simple process leads to a randomized round complexity of $O(\log f + \log_f \Delta + \operatorname{poly}\log\log n)$ by applying any MIS algorithm that runs in $O(\log \Delta + \operatorname{poly}\log\log n)$ rounds [25, 28] on the union of all subgraphs, which have no conflicts by construction. Then, $f = 2^{\sqrt{\log \Delta}}$ is chosen to achieve a runtime of $O(\sqrt{\log \Delta} + \operatorname{poly}\log\log n)$.

Deterministic 2-Ruling Set Sparsification. Our goal is to replace the above randomized sampling with a deterministic sampling that returns a subgraph G' with the same properties as those returned by the above construction [34, 33]. We slightly alter the sampling guarantees to allow for a relaxed maximum degree in G' of up to $\operatorname{poly}(f)$ instead of $O(f \log n)$. Instead of sampling each vertex with probability $f \cdot \log n/\Delta$ randomly and independently in a single round, we sample them in a deterministic manner in $O(\log\log \Delta)$ rounds. The way in which we design this deterministic sampling step is explained next.

The standard approach is to limit the randomness by sampling vertices using a carefully selected k-wise independent hash function. A naive implementation that samples vertices with probability $\frac{\operatorname{poly}(f)}{\Delta}$ would need a family of k-wise independent hash functions with $k = \Omega(\log_f n)$, since each vertex has $\operatorname{poly}(f)$ expected sampled neighbors. The need for $\Omega(\log_f n)$-wise independence results in a seed of length $\Omega(\log_f n \cdot \log \Delta)$. Since in $O(1)$ MPC rounds only $O(\log n)$ bits can be fixed, this one-step process appears to require $\Omega(\frac{\log \Delta}{\log f})$ many rounds[1], which is very far from being sublogarithmic.

Our approach to make this construction randomness-efficient relies on breaking down the sampling process into $O(\log\log \Delta)$ sub-sampling processes, each of which has weaker guarantees but requires only $O(1)$ rounds. In particular, the basis of our process is a simple, deterministic, constant-round routine that decreases the maximum degree by a $O(\sqrt{\Delta})$-factor, while ensuring that the maximum-to-minimum degree ratio of $O(f)$ is maintained, i.e., each vertex v has degree roughly $|N_G(v)|/\sqrt{\Delta}$ in G'.

Then, we repeatedly apply this degree-reduction routine to sparsify the neighborhoods of high-degree vertices until their degree drops to $2^{O(\log f)}$. It is easy to see that this requires at most $O(\log\log \Delta)$ repetitions. However, in each iteration, some downsampled neighborhoods may deviate from their expectation, say by an ϵ-factor. Such deviation is amplified each time, resulting in a potential error of $\epsilon^{O(\log\log \Delta)}$. Nevertheless, through a suitable f and ϵ, we can minimize the error and show that the subgraph G' has $\operatorname{poly}(f)$ maximum degree. Therefore, we can iterate through the $O(\log_f \Delta)$ degree classes (as in the randomized case) and apply our deterministic degree reduction to achieve the same result, up to a $O(\log\log \Delta)$ factor.

Further Comparison. Several sparsifications for MIS and 2-ruling sets in LOCAL and low-memory MPC have been studied. We include a brief comparison with the works of [18, 39, 33].

A deterministic $O(1)$-round sampling process appeared in the MIS algorithm of [18]. There, the goal is to reduce the maximum degree to at most n^ϵ while ensuring that the resulting subgraph maintains enough edges and the distribution of degrees is still representative of the original graph. They decrease the maximum degree by an $n^{\Omega(1)}$-factor for $O(1)$ times, until the desired bound is achieved. Since the expected new maximum degree is still on the order

[1] Here, shortening the seed length using a family of ε-approximate k-wise independent hash functions still requires $\omega(1)$ MPC rounds.

of $n^{\Omega(1)}$, concentration around the expectation can be achieved with $O(1)$-wise independence, and thus derandomized in $O(1)$ rounds. In contrast, in 2-ruling set, the main challenge is to subsample the neighborhoods of nodes with degree $d \ll n^{\Omega(1)}$. In fact, applying a similar subsampling method would require $\Omega(\log_d n)$-wise independence and $\Omega(\frac{\log \Delta}{\log f})$ rounds, as explained in the paragraph above. Thus, while the method in [18] is effective for high-degree nodes with $d = n^{\Omega(1)}$, handling smaller degrees requires a different approach.

The ruling set algorithm of [39] introduces a CONGEST sparsification that runs in $O(\log^2 n)$ rounds and deals with $O(\log \Delta)$ degree classes. There, a single sampling step requires a seed of length $O(\log^2 n)$ as they require guarantees stricter than ours. Specifically, their sparsification must maintain a low diameter and ensure proper coverage. Although their derandomization is CONGEST-efficient, it would require $O(\log n)$ MPC rounds, making it unsuitable to our setting.

Finally, we note that the faster randomized 2-ruling set algorithm of [33] relies on (informally) performing graph exponentiation on a sparsified subgraph. This approach relies on fixing the randomness of future iterations in advance, which simplifies the process of speeding up algorithms in LOCAL. The main challenge in adapting this approach to a deterministic setting is that existing techniques are generally effective at derandomizing only $O(1)$ steps of an algorithm. They do not easily extend to derandomize algorithms that simulate $\Omega(1)$ randomized rounds locally on each single machine via graph exponentiation. Consequently, achieving the same speed up deterministically appears to require a novel approach.

2 Preliminaries

In our analyses, we will use the notation $\text{poly}(\cdot)$ to refer to $(\cdot)^c$, for a constant $c > 0$ at the exponent that can be made arbitrarily large without affecting asymptotic bounds.

Primitives in MPC. We recall that basic computations can be performed in the MPC model with strongly sublinear local memory in $O(1)$ rounds deterministically [29, 30].

Therefore, tasks such as computing the degree of each vertex, ensuring neighborhoods of all vertices are stored on single machines, and collecting certain subgraphs onto a single machine will be used as black-box tools.

Derandomization Framework. A rich and successful line of research has studied the derandomization of algorithms in the parallel and distributed setting. In the MPC model, classic derandomization schemes using limited independence and the method of conditional expectation [38, 41], can be augmented with the power of local computation and global communication to achieve the expected result in $O(1)$ rounds.

We will often use the concepts of k-wise independence and family of k-wise independent hash functions (see, e.g., [40, 44]). Given a randomized process that works under k-wise independence, it is known how to construct a k-wise independent family of hash functions.

▶ **Lemma 3** ([1, 14, 21]). *For every $N, k, \ell \in \mathbb{N}$, there is a family of k-wise independent hash functions $\mathcal{H} = \{h : [N] \to \{0,1\}^\ell\}$ such that choosing a uniformly random function h from $\mathcal{H}$ takes at most $k(\ell + \log N) + O(1)$ random bits, and evaluating a function from $\mathcal{H}$ takes time* $\text{poly}(\ell, \log N)$ *time.*

Moreover, to show concentration around the expected value under k-wise independence, we will use the following tail bound.

▶ **Lemma 4** (Lemma 2.3 of [8]). *Let $k \geq 4$ be an even integer. Let $X_1, \ldots, X_n$ be random variables taking values in $[0,1]$. Let $X = X_1 + \ldots + X_n$ denote their sum and let $\mu \leq \mathbb{E}[X]$ satisfying $\mu \geq k$. Then, for any $\epsilon > 0$, we have*

$$\Pr\left[|X - \mathbb{E}[X]| \geq \epsilon \cdot \mathbb{E}[X]\right] \leq 8\left(\frac{2k}{\epsilon^2 \mu}\right)^{k/2}.$$

We consider randomized algorithms that succeed in expectation when their random choices are made using a family of k-wise independent hash functions $\mathcal{H}$. Once our algorithm (randomly) picks a hash function h, then all choices are made deterministically according to h. Thus, our problem is that of deterministically finding a hash function that achieves a result as good as the expectation.

The by-now standard MPC derandomization process can be broken down into two parts: (i) show that the family of hash functions $\mathcal{H}$ has size poly(n) *and* produces the desired result in expectation, and (ii) find one good hash function by applying the method of conditional expectation in a distributed fashion. We will focus on establishing (i), since (ii) can then be achieved by known MPC derandomization methods introduced by earlier works [12, 15, 18] to which we refer for further details. It is worth mentioning that for step (ii) to be solved using earlier tools as a black-box, the aimed expectation should be expressed as a sum of locally computable quantities by each individual machine, i.e., the individual expectation of each node that a machine stores.

3 Deterministic 2-Ruling Set in Linear MPC

We first introduce the reader to several sets of nodes that play a crucial role in our algorithm. These sets of nodes are defined to reflect how a node will be handled by our algorithm. Specifically, the core of the algorithm is a downsampling procedure that outputs a sufficiently small subgraph on which we will compute a maximal independent set with the goal of *ruling* a large fraction of nodes in the original graph.

Observe that if a node has a neighbor in the downsampled graph, then it will have some node in the maximal independent set at distance at most two. This means that if a node is likely to have a sampled neighbor, then it is likely to be ruled, and we call such a node *good*. In the following, our definitions and algorithm are parameterized by a constant $\varepsilon = 1/40$, which has not been optimized.

▶ **Definition 5** (Good Node). *A node $v \in G$ is good if it satisfies $\sum_{u \in N(v)} \frac{1}{\sqrt{\deg(u)}} \geq \deg(v)^{\varepsilon}$.*

If a node v is not good, i.e., $\sum_{u \in N(v)} \frac{1}{\sqrt{\deg(u)}} < \deg(v)^{\varepsilon}$, then we say that v is a *bad* node. Bad nodes are split into $O(\log \Delta)$ degree classes as follows. Let d_0 be a sufficiently large constant and $d_{\max} = \lceil \log \Delta \rceil$.

▶ **Definition 6** (Bad Node Classes). *For $d \in \{2^{d_0}, 2^{d_0+1}, \ldots, 2^{d_{max}}]$, the set B_d includes all bad nodes with degree in $[d, 2d)$.*

Therefore, bad nodes are likely to have few sampled nodes. This fact motivates the following observation. If a (bad) node has *many* bad nodes within its 2-hop neighborhood, then it is likely that at least one of such bad ones is in the maximal independent set. If that is the case, we call such nodes *lucky* bad nodes, as specified in the following definition.

▶ **Definition 7** (Lucky Bad Nodes). *For $d \in \{2^{d_0}, 2^{d_0+1}, \ldots, 2^{d_{max}}]$, the set $\overline{B}_d \subseteq B_d$ includes each node $u \in B_d$ such that u has a neighbor w with $|N(w) \cap B_d| \geq 6d^{0.6}$. If there are multiple such w's, pick one arbitrarily and let S_u be an arbitrarily chosen subset of $N(w) \cap B_d$ such that $|S_u| = 6d^{0.6}$.*

With these definitions in mind, we are now ready to present our deterministic constant-round 2-Ruling Set algorithm in the linear regime of MPC.

The algorithm operates in three simple steps: Sampling, Gathering, and MIS Computation. The first step of the algorithm samples each node v with probability $\deg^{-1/2}(v)$. The sampling probability is chosen to ensure that the downsampled graph has a linear number of edges. Moreover, we will slightly alter the downsampled graph to include all nodes that do not satisfy certain requirements, without affecting the asymptotic size of this subgraph. Therefore, in the second step, we will be able to collect such subgraph onto a single machine. Then, the MIS computation begins by running one iteration of Luby's MIS on (part of) the subgraph from the previous step and continues by extending such independent set to a maximal one locally.

We will prove several desirable properties about the three-step algorithm above that lead to a reduction of a $d^{\Omega(1)}$-factor for each degree class d. Therefore, by repeating this three-step algorithm $O(1)$ times, the number of edges over all degree classes converges to $O(n)$ and thus can be collected and solved locally, completing the proof of Theorem 1.

Next, we present the algorithm in more detail and then proceed to analyzing its three steps with a particular focus on randomness efficiency. In fact, such randomness-efficient analyses will allow for a simple derandomization.

3.1 The Algorithm

Sampling Step. Let $G = (V, E)$ be the input graph with n vertices and m edges. Let V_{samp} denote the set of sampled vertices. We include each vertex $v \in V$ in V_{samp} with probability $p_v = \frac{1}{\sqrt{\deg(v)}}$, according to a family of k-wise independent random variables with $k = O(1)$.

Gathering Step. We gather several subsets of nodes whose (combined) induced subgraph will be shown to have a linear number of edges. Gathered nodes are those either sampled in the previous step or not satisfying certain properties as formally defined below. Let V^* denote the union of the following node subsets, which are being gathered locally onto a single machine:

1. The set of sampled nodes V_{samp};
2. Every good node that is *not sampled* and has *no sampled neighbors*;
3. For each d, every lucky bad node $u \in \overline{B_d}$ that has either less than $d^{0.1}$ sampled nodes in S_u or one of the sampled nodes in S_u has more than $d^{2\varepsilon}$ sampled neighbors; as formalized in Lemma 10.

MIS Computation. Our goal is now to compute a maximal independent set on the locally gathered subgraph $G[V^*]$ to rule all but roughly at most a $\Delta^{\Omega(1)}$-fraction of nodes in G. We achieve this by first computing a partial MIS on the sampled bad vertices, i.e., $\bigcup_d B_d \cap V_{\text{samp}}$, using a variation of Luby's algorithm as detailed in the proof of Lemma 12. Afterward, we can simply compute an MIS locally (and thus sequentially) on the remaining vertices, which are not incident to the partial MIS computed earlier.

Output Properties. We expect that the output given by the *derandomization* of the above three-step process satisfies the following properties. We will later use these properties to achieve a deterministic constant-round complexity. Observe that we can ignore constant-degree nodes since they can be gathered and dealt with locally at last.

- **Good nodes:** All good nodes in G are ruled after the MIS step.

- **Uncovered lucky bad nodes:** For each d, after the computation of a partial MIS, only a $d^{\Omega(1)}$-fraction of lucky bad nodes remains uncovered.
- **Uncovered bad nodes:** For each d, the number of bad nodes in $B_d \setminus \overline{B}_d$ is only a $d^{\Omega(1)}$-fraction of all nodes with initial degree at least d in G .

3.2 Analysis

We first establish that good nodes are likely to have a neighbor in V_{samp}. Since we will compute an MIS on $V^* \supseteq V_{samp}$, such good nodes will be at distance at most 2 from a node in the MIS. Moreover, good nodes that have no sampled neighbor will be shown to be incident to a linear number of edges, allowing us to gather them as part of V^*.

▶ **Lemma 8.** *Every good vertex v has a neighbor in V_{samp} with probability at least $1 - \frac{1}{poly(deg(v))}$.*

Proof. For any vertex u, let X_u be the indicator random variable for the event $u \in V_{\text{samp}}$, and X be the random number of neighbors of v in V_{samp}. Further, let $\mu := \mathbb{E}[X] = \sum_{u\in N(v)} \mathbb{E}[X_u] = \sum_{u\in N(v)} \Pr[X_u = 1] \geq \deg(v)^{\varepsilon} \gg k$, since nodes of constant degree can be ignored and dealt with separately at last by collecting them onto a single machine. By applying Lemma 4, we have

$$\Pr[X = 0] \leq \Pr[|X - \mu| \geq \mu] \leq 8 \cdot \left(\frac{k\mu + k^2}{\mu^2}\right)^{k/2} \leq 8 \cdot \left(\frac{2k}{\mu}\right)^{k/2} = \frac{1}{\text{poly}(\deg(v))},$$

which proves the lemma. ◀

Toward the goal of ruling lucky bad nodes, we next show that bad nodes are likely to have few sampled neighbors. This means that sampled bad nodes, by having a low degree in the sampled graph, will have higher chances of being in the partial MIS computed later.

▶ **Lemma 9.** *Any node $u \in B_d$ has at most $d^{2\varepsilon}$ sampled neighbors with probability at least $1 - \frac{1}{poly(d)}$.*

Proof. Recall that for any $u \in B_d$, it holds that $\sum_{w\in N(u)} \frac{1}{\sqrt{\deg(w)}} < \deg(u)^{\varepsilon}$. We will use this fact to prove that the number of sampled neighbors does not deviate by more than $O(d^{2\varepsilon})$ with probability at least $1 - \frac{1}{\text{poly}(d)}$. Let X_w be the indicator random variable for the event $w \in V_{\text{samp}}$, and X be the random number of neighbors of u in V_{samp}. Let $\mu = \mathbb{E}[X] = \sum_{w\in N(u)} \mathbb{E}[X_w] = \sum_{w\in N(u)} \Pr[X_w = 1] < \deg(u)^{\varepsilon} < 2d^{\varepsilon}$. By applying Lemma 4, we get

$$\Pr[|X - \mu| \geq d^{2\varepsilon} - \mu] \leq 8 \cdot \left(\frac{k^2 + k\mu}{(d^{2\varepsilon} - \mu)^2}\right)^{k/2} \leq 8 \cdot \left(\frac{2k^2}{d^{\varepsilon}}\right)^{k/2} = \frac{1}{\text{poly}(d)}.$$

Note that for small values of d, our constant d_0 can be chosen such that $2^{d_0 \cdot \varepsilon} = \Omega(k^2)$. ◀

The next lemma proves that each lucky bad node u has a large number of nodes sampled out of its set S_u. Specifically, we need to show that the number of sampled nodes in S_u is higher than the degree of such nodes in the sampled graph. This fact will be used to ensure that lucky bad nodes have a vertex, within their 2-hop neighborhoods, in the MIS, thereby, ensuring their coverage.

▶ **Lemma 10.** *For any lucky bad node u, its set $S_u \subseteq B_d$ of cardinality $6d^{0.6}$ contains at least $d^{0.1}$ sampled nodes and each sampled node in S_u has at most $d^{2\varepsilon}$ sampled neighbors with probability at least $1 - \frac{1}{poly(d)}$.*

Proof. By Lemma 9 and a union bound over the set S_u of $6d^{0.6}$ nodes, none of them has more than $d^{2\varepsilon}$ sampled neighbors with probability at least $1 - \frac{1}{\text{poly}(d)}$. Our goal is now to prove that the number of sampled vertices within S_u is less than $d^{0.1}$ with probability at most $\frac{1}{\text{poly}(\deg(u))} = \frac{1}{\text{poly}(d)}$.

Let X be the random number of sampled vertices in S_u, and let $\mu = \mathbb{E}[X] \geq 3d^{0.1}$, since each vertex in B_d is sampled with probability at least $1/\sqrt{2d}$. By applying Lemma 4, the probability of X deviating by more than $d^{0.1}$ from its expected value is

$$\Pr[|X-\mu| \geq \mu - d^{0.1}] \leq 8 \cdot \left(\frac{2k\mu}{(\mu - d^{0.1})^2}\right)^{k/2} \leq 8 \cdot \left(\frac{2k}{d^{0.1}}\right)^{k/2} = \frac{1}{\text{poly}(\deg(u))}. \qquad ◀$$

We now use the above lemmas, together with a bound on the number of edges induced by the sampling step, to prove that our gathering step effectively collects $O(n)$ edges.

▶ **Lemma 11.** *The subgraph induced by $G[V^*]$ has $O(n)$ edges in expectation.*

Proof. Our goal is to prove that the expected sum of the original degrees of nodes in V^* is $O(n)$, which clearly upper bounds the number of edges in the induced subgraph. To do so, we analyze each subset individually.

We first analyze the expected number of edges induced by V_{samp}. Let X denote the random number of edges within the subgraph $G[V_{\text{samp}}]$. Let Y_e be an indicator random variable for the event that edge e is in $G[V_{\text{samp}}]$. To aid our analysis, we orient each edge in the graph from the endpoint with lower degree to the endpoint with higher degree. Now, consider an edge $e = (u, v)$ with $\deg(u) \leq \deg(v)$. Vertices u and v are each sampled with probability at most $\frac{1}{\sqrt{\deg(u)}}$. By pairwise independence, the probability of edge e being in $G[V_{\text{samp}}]$ is bounded by $\frac{1}{\deg(u)}$. Consequently, the expected number of edges is $\mathbb{E}[X] = \sum_{v \in V} \sum_{e \in \text{out}(v)} \mathbb{E}[Y_e] \leq \sum_{v \in V} \sum_{e \in \text{out}(v)} \frac{1}{\deg(u)} = O(n)$.

Next, let $\overline{V}_{\text{good}}$ denote the set of good nodes that have no sampled neighbor and Y the random number of edges incident to $\overline{V}_{\text{good}}$ in G. By Lemma 8, each good node v is in $\overline{V}_{\text{good}}$ with probability at most $1/\text{poly}(\deg(v))$. Thus,

$$\mathbb{E}[Y] \leq \sum_{v \in V} \deg(v) \cdot \Pr[v \in \overline{V}_{\text{good}}] \leq \sum_{v \in V} \frac{\deg(v)}{\text{poly}(\deg(v))} = O(n).$$

Finally, let the set $B'_d \subseteq \overline{B_d}$ include each unlucky bad node u such that either less than $d^{0.1}$ vertices in S_u are sampled or any sampled node in S_u has more than $2d^{\varepsilon}$ sampled neighbors. By Lemma 10, each node u is in B'_d with probability at most $1/\text{poly}(d)$. Let Z be the random number of edges incident to B'_d. We have

$$\mathbb{E}[Z] \leq \sum_{i=d_0}^{d_{\max}} \sum_{u \in B_{2^i}} \deg(u) \cdot \Pr[u \in B'_{2^i}] \leq \sum_{i=d_0}^{d_{\max}} \sum_{u \in B_{2^i}} \frac{2d}{\text{poly}(d)} \leq \sum_{i=d_0}^{d_{\max}} |B_{2^i}| = O(n). \qquad ◀$$

Derandomize Sampling and Gathering Steps. We are now ready to discuss how the above Sampling and Gathering steps can be turned into a deterministic linear MPC algorithm. Recall that each vertex is sampled according to a family of k-wise independent random variables with $k = O(1)$. A family $\mathcal{H}$ of k-wise independent hash functions such that $h \in \mathcal{H} : [n] \to [n^3]$ can be specified using a random seed of length $O(\log n)$, meaning that $|\mathcal{H}| = \text{poly}(n)$. Each h maps the n vertex IDs (assumed to be from 1 up to n) to an integer in $[n^3]$. Then, each vertex is sampled and belongs to V_{samp} iff its ID is mapped to an integer that is at most $\left\lfloor n^3/\sqrt{deg(v)} \right\rfloor$ with respect to h, where the floor affects results only asymptotically. Each vertex can now locally check whether it will be included in V^* for a specified hash function h. In fact, the machine that v is assigned to stores all v's neighbors and the set S_v if v is a lucky bad node. Therefore, it is easy to see that each node can computed the objective function $|E(G[V^*])|$ locally, and we can thus apply the distributed method of conditional expectation. Since $|\mathcal{H}| = \text{poly}(n)$, after a constant number of rounds we will find a h that ensures $|E(G[V^*])| = O(n)$.

We now turn to analyzing the MIS step. Recall that we first compute a partial MIS on the sampled bad nodes in order to rule all but a small fraction of lucky bad nodes. The next lemma explains how such an independent set is being computed.

▶ **Lemma 12.** *Let $\hat{B}_d$ include each node $u \in \overline{B_d}$ that satisfies the property of Lemma 10. After the partial MIS computation, each node $u \in \hat{B}_d$ will be ruled with probability at least $1 - \frac{45}{d^\varepsilon}$ for all $d \in [d_0, d_{max}]$. This result depends only on the randomness used in the MIS computation.*

The proof of Lemma 12 is provided in Appendix A.

The above lemma turns out not to be sufficient to derandomize our MIS step. In fact, we need to show that all degree classes of lucky bad nodes have a high enough chance of being ruled *simultaneously*. This is due to the fact that in the derandomization process, we can control only *one* objective function and not $O(\log \Delta)$ as the number of degree classes would appear to require. In the next lemma, we show how to define a pessimistic estimator that solves this issue.

▶ **Lemma 13.** *After the partial MIS computation, all but at most $\frac{|\overline{B}_d|}{d^{\Omega(1)}}$ nodes will be ruled in expectation, for all d simultaneously.*

Proof. Let us first reason about a fixed d and then about all d's simultaneously.

Recall that $\hat{B}_d$ include each node $u \in \overline{B_d}$ that satisfies the property of Lemma 10. There are at most $\frac{|\overline{B}_d|}{\text{poly}(d)}$ vertices in $\overline{B}_d \setminus \hat{B}_d$ by Lemma 10. Then, any vertex in $\hat{B}_d$ is ruled with probability at least $1 - \frac{45}{d^\varepsilon}$ by Lemma 12. Therefore, by linearity of expectation, the number of non-ruled vertices in is at most $45|\overline{B}_d|/d^\varepsilon$.

Our goal is now to define a *single* objective function whose expected value ensures that the same asymptotic result holds for *all* d simultaneously. Let X_d be the random number of unruled nodes in $\overline{B}_d$, for each d. We define our objective function Q, which will serve as a "pessimistic estimator", as a weighted sum of the X_d's as follows.

$$Q = \sum_{i=d_0}^{d_{\max}} X_{2^i} \cdot \frac{2^{i\cdot\frac{\varepsilon}{2}}}{|\overline{B}_{2^i}|},$$

so that we get

$$\mathbb{E}[Q] = \sum_{i=d_0}^{d_{\max}} \mathbb{E}[X_{2^i}] \cdot \frac{2^{i\cdot\frac{\varepsilon}{2}}}{|\overline{B}_{2^i}|} \le \sum_{i=d_0}^{d_{\max}} \frac{45|\overline{B}_{2^i}|}{2^{i\varepsilon}} \cdot \frac{2^{i\cdot\frac{\varepsilon}{2}}}{|\overline{B}_{2^i}|} = \sum_{i=d_0}^{d_{\max}} \frac{45}{2^{i\varepsilon/2}} = O(1),$$

where the convergency follows from choosing a sufficiently large constant $d_0 = O(\varepsilon^{-1})$. Observe that the expected value of Q ensures that, for each set $\overline{B}_d$, the number of nodes which are not ruled after running our Luby's step is $X_d \le \mathbb{E}[Q] \cdot \frac{|\overline{B}_d|}{d^{\varepsilon/2}} = \frac{|\overline{B}_d|}{d^{\Omega(1)}}$. ◀

Deterministic MIS Step. We now present an efficient derandomization of the above partial MIS computation in the linear MPC regime. As discussed in Lemma 12, our family $\mathcal{H}$ of pairwise independent hash functions has size $|\mathcal{H}| = \text{poly}(n)$. Note that each lucky bad node u can store in a single machine its set S_u and all of their sampled neighbors since $|S_u| \cdot d^{2\varepsilon} = O(d) = O(deg(u))$. Then, each vertex u can check whether it will be ruled under a specified hash function h. Therefore, we can compute u's contribution to $Q(h)$ locally, where $Q(h)$ is the objective function of Lemma 13 under a specified hash function h. This allows us to apply the distributed method of conditional expectation with objective Q to find a good hash function with $Q(h) = O(1)$ in a constant number of rounds.

Counting the bad nodes. Let $V_{\geq d}$ denote the set of all nodes in G with initial degree at least d, and let the set $B_d^* \overset{\text{def}}{=} B_d \setminus \overline{B}_d$. It remains to prove that the set B_d^* contains only a small fraction of nodes. The next lemma is equivalent to Lemma 9 of [11] up to some parameters change.

▶ **Lemma 14.** *For any degree $d \in [2^{d_0}, 2^{d_{max}}]$, we have that $|B_d^*| \leq 12|V_{\geq d}|/d^{0.4}$.*

Proof. For a bad node v, it is easy to see by contradiction that at least $d/2$ of v's neighbors have degree at least $d^{2(1-\varepsilon)}/4$ (see also Lemma 8 of [11]). Let $d' = \frac{d^{2(1-\varepsilon)}}{4}$. Therefore, any node $v \in B_d^*$ has at least $d/2$ neighbors in $V_{\geq d'}$. Furthermore, any node in $V_{\geq d'}$ neighboring a node in B_d^* has at most $6d^{0.6}$ edges connecting to nodes in $B_d \supseteq B_d^*$. As a result of these observations, we derive the following inequality:

$$d/2 \cdot |B_d^*| \leq 6|V_{\geq d'}| \cdot d^{0.6},$$

which together with the fact that $d' \geq d$, for d large enough, proves the lemma. ◀

Bounding Total Runtime. In the above paragraphs, we showed how to achieve deterministically the properties required by our three-step algorithm outlined at the beginning of this section. We now rove that repeating this process $O(1)$ times reduces the size of the graph to $O(n/\Delta)$, implying that the remaining nodes can be collected and solved for locally.

▶ **Lemma 15.** *At the end of the first iteration, the number of remaining uncovered vertices with degree at least d, denoted by $V_{\geq d}^{(1)}$, satisfies*

$$|V_{\geq d}^{(1)}| \leq |V_{\geq d}|/d^{\varepsilon'}.$$

Proof. The remaining uncovered vertices are only bad nodes. An uncovered bad node of degree $[d, 2d)$ can be either in B_d^* (Lemma 14) or remained uncovered after running the deterministic MIS step (Lemma 13). Over all $d, \ldots, 2^{d_{\max}}$, this leads to:

$$|V_{\geq d}^{(1)}| \leq \sum_{i=\log d}^{d_{\max}} |B_{2^i}^*| + \frac{|\overline{B}_d|}{2^{\Omega(i)}} \leq \sum_{i=\log d}^{d_{\max}} \frac{12|V_{\geq 2^i}|}{2^{0.4 \cdot i}} + \frac{|\overline{B}_d|}{2^{\Omega(i)}} \leq |V_{\geq d}| \sum_{i=\log d}^{d_{\max}} \frac{1}{2^{\Omega(i)}} = \frac{|V_{\geq d}|}{d^{\Omega(1)}},$$

where the last inequality follows from $|\overline{B}_d| \leq |V_{\geq d}|$, and the final bound is due to the geometric sum being asymptotically dominated by the first term. ◀

Having established, in Lemma 15, the progress made at each iteration by our three-step process, we can now apply a simple induction to show the desired bound on the progress made after several iterations.

▶ **Lemma 16.** *After $O(1)$ iterations, the graph induced by uncovered nodes has $O(n)$ edges.*

Proof. Let $V^{(k)}_{\geq d}$ denote the number of remaining uncovered vertices with degree at least d at iteration k. Our goal is to prove that after k iterations, it holds that $V^{(k)}_{\geq d} \leq V_{\geq d}/d^{k\varepsilon'}$ so that for $k = O(1/\varepsilon')$, we get $V^{(k)}_{\geq d} \leq V_{\geq d}/d^{1.1}$. The base case for $k = 1$ follows from Lemma 15. Now, let us assume that $V^{(k-1)}_{\geq d} \leq V_{\geq d}/d^{(k-1)\varepsilon'}$. By a straightforward application of Lemma 15, we have that $V^{(k)}_{\geq d} \leq |V^{(k)}_{\geq d}|/d^{\varepsilon'} \leq V_{\geq d}/d^{k\varepsilon'}$, as desired. Now, since the number of nodes with degree $[d, 2d)$ is upper bounded by $|V_{\geq d}|$, the total number of edges is at most $\sum_{i=\log d_0}^{\log d_{\max}} V_{\geq d} \cdot 2^{i+1-1.1\cdot i} = \sum_{i=\log d_0}^{\log d_{\max}} O(n/2^{0.1\cdot i}) = O(n)$. ◀

4 Deterministic 2-Ruling Set in Sublinear MPC

In this section, we show that for an input graph with maximum degree Δ, a 2-ruling set can be computed deterministically in the strongly sublinear memory regime of MPC in $\tilde{O}(\log^{1/2} n)$ rounds.

We start by introducing a simple, deterministic, constant-round routine that reduces the size of each high-degree neighborhood by a $\sqrt{\Delta}$-factor, where high-degree refers to node with degree at least $\log(n) \cdot \Delta^{0.6}$. For ease of exposition, assume that high-degree vertices form a set U, and that V is the set of all vertices (including high-degree vertices) that are being downsampled. Therefore, we reason about a bipartite graph $G = (U \sqcup V, E)$, where each node in $u \in U$ is connected to each vertex $v \in N_G(u)$ in the other part. Our goal is to ensure that each vertex u has roughly $N_G(u)/\sqrt{\Delta}$ neighbors deterministically. For simplicity, in the next lemma, we make two assumptions: (i) the neighbors of each vertex fit into a single machine, and defer the other case to Lemma 18; (ii) we are given a certain coloring of G that we discuss how to achieve at the end of this section.

▶ **Lemma 17.** *Let G be a graph with bipartition $V(G) = U \sqcup V$ and Δ be an upper bound on the maximum degree of any node in U such that $\Delta \in O(n^\alpha)$ for some $\alpha < 1$. Furthermore, assume that each node in V is given a color out of a palette of $O(\Delta^6)$ colors, such that any two distinct nodes $v, v' \in V$ that have a common neighbor in U are assigned distinct colors. Then, there exists a deterministic constant-round sublinear MPC algorithm that computes a subset $V^{sub} \subseteq V$ such that for any node $u \in U$ with $deg_G(u) \geq \log(n) \cdot \Delta^{0.6}$, it holds that $|N_G(u) \cap V^{sub}| \in \left[\frac{1}{3\sqrt{\Delta}}|N_G(u)|, \frac{1}{\sqrt{\Delta}}|N_G(u)|\right]$. The global space usage is linear in the input size.*

Proof. Let us assume that each node $v \in V$ knows its own color c_v of a coloring satisfying the above properties. Then, nodes in V apply a hash function h from a k-wise independent family $\mathcal{H}$ that maps each color to an integer in $[\lceil 3\sqrt{\Delta}/2 \rceil]$. A node v is then sampled under h iff $h(v) = 1$, which occurs with probability $1/\lceil 3\sqrt{\Delta}/2 \rceil$, where the ceil affects our results only asymptotically. We choose $k = 4c \log_\Delta n$, for constant $c > 0$, so that the seed length to select a hash function from $\mathcal{H}$ is at most $\ell = O(\log_\Delta n) \cdot \max\{O(\log \Delta^6), O(\log \sqrt{\Delta})\} = O(\log n)$, i.e., the family $\mathcal{H}$ has size $\mathrm{poly}(n)$.

We prove that for each vertex $u \in U$ with degree larger than $\log n \cdot \Delta^{0.6}$, the probability of having between $\frac{1}{3\sqrt{\Delta}}|N(u)|$ and $|N(u)|/\sqrt{\Delta}$ neighbors within V^{sub} is at least $1 - \frac{1}{n^c}$, i.e., the count of v's neighbors in V^{sub} deviates by at most $\frac{1}{3\sqrt{\Delta}}|N(u)|$. For each neighbor v of u, let X_v be an indicator random variable for the event $v \in V^{sub}$. Define $X = \sum_{v \in N(u)} X_v$ as the number of neighbors of u in V^{sub}. Then, $\mu = \mathbb{E}[X] = \frac{2|N(u)|}{3\sqrt{\Delta}} \geq c \log n \Delta^{0.1}$. By applying

Lemma 4, we have:

$$\Pr[|X-\mu| \geq \mu/2] \leq 8\left(\frac{4k\mu+4k^2}{\mu^2}\right)^{k/2} \leq 8\left(\frac{16c^2\Delta^{0.1}\log^2 n+32c^2\log^2 n}{\Delta^{0.2}c^2\log^2 n}\right)^{k/2}$$
$$\leq 8\left(\frac{1}{\Delta^{0.1}}\right)^{\frac{4c}{2}\cdot\frac{\log n}{\log\Delta}} \leq \frac{1}{n^{2c}}.$$

Therefore, the expected number of high-degree vertices in U whose count of sampled neighbors deviates by more than $\mu/2$ is at most $n^{2c-1} < 1$. This means that we can apply the method of conditional expectation in a distributed fashion with as objective function the number of bad nodes, i.e., those whose sampled neighborhood deviates from the expectation by more than half. Since the memory capacity of each machine is $O(n^\alpha)$, each machine can compute locally the contribution to the objective of all the vertices (and their neighbors) it stores. Therefore, after $O(1)$ rounds, we find a hash function such that *all high-degree vertices* in U have the desired number of sampled neighbors. ◀

Next, we discuss how to extend Lemma 17 to handle the case in which not all neighbors of a vertex in U can be collected onto a single machine. In particular, if $\Delta \gg n^\alpha$, then aiming for a reduction of a $\sqrt{\Delta}$-factor might not be viable, given the constrained local memory. Due to that, we slightly relax our goal and reduce our high-degree neighborhoods by a n^ε-factor, for some constant $\varepsilon < \alpha$. To achieve that, we split edges into groups so that each machine is assigned $n^{c\cdot\varepsilon}$ edges, for $c > 1$. While we can only control the deviation of each single group of edges, we will be able to bound the overall number of neighbors, i.e., edges per node, using the fact that there are at most $\Delta/n^{c\cdot\varepsilon}$ groups.

▶ **Lemma 18.** *Let G be a graph with bipartition $V(G) = U \sqcup V$. Let Δ be an upper bound on the maximum degree of any node in U such that $\Delta \geq n^{10\varepsilon}$, for some constant $\varepsilon > 0$. Then, there exists a deterministic constant-round sublinear MPC algorithm that computes a subset $V^{sub} \subseteq V$ such that for any node $u \in U$ with $deg_G(u) \geq \log(n)\cdot\Delta^{0.6}$, it holds that $|N_G(u) \cap V^{sub}| \in \left[\frac{1}{2n^\varepsilon}|N_G(u)|, \frac{3}{2n^\varepsilon}|N_G(u)|\right]$. The global space usage is linear in the input size.*

Proof. Consider an arbitrary vertex $u \in U$ with degree at least $\log(n)\cdot\Delta^{0.6}$. The idea is to split edges of u into groups of size at most $n^{4\varepsilon}$, which fits into the memory of one machine. Specifically, each machine holds $n^{4\varepsilon}$ edges except for a single machine that holds any remaining edges, which are at most $n^{4\varepsilon}$. Then, we sample nodes in V with probability $n^{-\varepsilon}$ according to a family of $O(1)$-wise independent hash function. Using a calculation similar to that of Lemma 17, we can find a hash function such that all groups of $n^{4\varepsilon}$ edges have $n^{3\varepsilon} \pm n^{2\varepsilon}$ sampled edges. Then, the total number of sampled neighbors is at least

$$\sum_{\text{machine } i} n^{3\varepsilon} - n^{2\varepsilon} \geq \left\lfloor\frac{|N_G(u)|}{n^{4\varepsilon}}\right\rfloor\cdot\left(n^{3\varepsilon}-n^{2\varepsilon}\right) \geq \frac{|N_G(u)|}{n^\varepsilon} - \frac{|N_G(u)|}{n^{2\varepsilon}} - n^{3\varepsilon} \geq \frac{|N_G(u)|}{2n^\varepsilon},$$

where $n^{3\varepsilon} = o(\frac{|N_G(u)|}{2n^\varepsilon})$ since $N_G(u) \geq n^{6\varepsilon}$. An analogous calculation shows that the total number of sampled neighbors for any vertex u is at most $\frac{3|N_G(u)|}{2n^\varepsilon}$. ◀

We are now ready to present our $O(\log\log\Delta)$ sparsification. We show that we can find a subset of nodes incident to all nodes in U such that their induced maximum degree is $2^{O(\log f)}$ for $f = 2^{\sqrt{\log\Delta}}$. This is achieved by repeating the sampling processes of Lemmas 17 and 18 for $O(\log\log\Delta)$ times. Here, one key observation to bound the deviation is that in each run of Lemma 17 only the lower tail may deviate up to a 1/3-factor from $\frac{|N_G(u)|}{\sqrt{\Delta}}$. So, the final multiplicative error will be $3^{O(\log\log\Delta)} = \operatorname{poly}\log\Delta$.

▶ **Lemma 19.** *Let G be a graph with bipartition $V(G) = U \sqcup V$. Let Δ and $\frac{\Delta}{f}$ be an upper bound on the maximum degree and a lower bound on the minimum degree, respectively, of any node in U for any parameter $f \leq \frac{\Delta^{0.4}}{\log n}$ and $f \geq \mathrm{poly}(\log n)$. There exists a sublinear MPC algorithm that computes in $O(\log\log\Delta)$ rounds a subset $V^{sub} \subseteq V$ such that for any node $u \in U$ with $\deg_G(u) \geq \frac{\Delta}{f}$, it holds that $|N_G(u) \cap V^{sub}| \in [1, 2^{O(\log f)}]$. The algorithm global space usage is linear in the input size.*

Proof. Our goal is to find a suitable set V^{sub} by applying the sparsification outlined in Lemma 17. If $\Delta \geq n^\alpha$, we first apply Lemma 18 for $O(1/\varepsilon) = O(1)$ times until the maximum degree in U is within the memory capacity of a single machine $O(n^\alpha)$, which can be achieved by setting $\varepsilon \leq \frac{\alpha}{10}$, i.e., $n^\alpha \geq n^{10\varepsilon}$. Define $\Delta' \leq n^\alpha$ as the maximum degree in U after downsampling vertices in V for $O(1)$ iterations as per Lemma 18. Notice that the minimum degree in U is now $c \cdot \frac{\Delta'}{f}$, for some constant $c > 0$. Then, we run the algorithm of Lemma 17 for $k = O(\log\log\Delta)$ iterations, and stop as soon as the minimum degree in U is within $2^{O(\log f)}$. We prove by induction that after k iterations nodes have degrees in

$$\left[\frac{c}{f \cdot 3^k}(\Delta')^{1/2^k}, (\Delta')^{1/2^k}\right].$$

The base case follows from Lemma 17. The induction step then follows from

$$\left[\frac{c}{f \cdot 3^{(k-1)}}(\Delta')^{1/2^{(k-1)}} \cdot \frac{1}{3(\Delta')^{1/2^k}}, (\Delta')^{1/2^{(k-1)}} \cdot \frac{1}{(\Delta')^{1/2^k}}\right] = \left[\frac{c}{f \cdot 3^k}(\Delta')^{1/2^k}, (\Delta')^{1/2^k}\right].$$

By choosing $k = \lfloor \log\log\Delta' - \log(2\log(f \cdot \log\Delta'))\rfloor$, one can verify that, for any vertex in U, the minimum degree in the downsampled graph will be at least one, and the maximum degree at most $2^{O(\log(f \cdot \log\Delta))} = 2^{O(\log f)}$. ◀

Our 2-ruling set algorithm is paramterized by $f = 2^{\sqrt{\log\Delta}}$. On a high-level, we mimic the randomized local 2-ruling set algorithm of [35]. In each iteration i, $0 \leq i \leq \lfloor \log f \rfloor$, we address the set of vertices with degree in $(\Delta/f^{i+1}, \Delta/f^i]$. We apply the sparsification of Lemma 19 on each set of high-degree vertices, one set at a time sequentially. Each sparsified subgraph is then put aside and, together with all incident nodes in G, is removed from further consideration before starting the next iteration. At the end, the union of all subgraphs of induced maximum degree $2^{O(\log f)}$ and possibly some remaining low-degree vertices are given in input to an MIS algorithm, whose solution is effectively a 2-ruling set. We detail the algorithm in the following pseudocode and proceed to its analysis below.

Algorithm 1 Sublinear 2-Ruling Set.

```
$f \leftarrow 2^{\sqrt{\log\Delta}}$; $M \leftarrow \emptyset$
for $i \leftarrow 0, 1, \cdots, \lfloor \log f \rfloor$ do
    $U \leftarrow \{v \in V \mid \deg_G(v) \in (\frac{\Delta}{f^{i+1}}, \frac{\Delta}{f^i}]\}$; $V' \leftarrow V$
    $G' \leftarrow (U \sqcup V', E' = \{(u,v) \mid u \in U, v \in V', (u,v) \in E\})$ ▷ Bipartition for sparsification
    for $j \leftarrow 1, 2, \cdots, O(\log\log\Delta)$ do                    ▷ See also Lemma 19
        $\Delta' \leftarrow$ maximum degree in $G'$
        $V' \leftarrow$ sample $v \in V'$ with prob. $\max\{\frac{2}{3\sqrt{\Delta'}}, \frac{1}{n^\varepsilon}\}$
    $M \leftarrow M \cup V'$
    $V \leftarrow V \setminus (V' \cup N_G(V'))$                    ▷ Remove neighbors of sampled set
Return MIS on $G[M \cup V]$
```

The proofs of the next two lemmas are fairly standard and deferred to Appendix B.

▶ **Lemma 20.** *At the end of iteration i, $1 \leq i \leq \lfloor \log f \rfloor$, all vertices still in V have degree at most* $\max\{\frac{\Delta}{f^i}, 2^{O(\log f)}\}$.

▶ **Lemma 21.** *After $\lfloor \log f \rfloor$ iterations, the subgraph induced by M together with vertices still in V, i..e, $G[M \cup V]$, has maximum degree $2^{O(\log f)}$.*

Proof of Theorem 2. As proved in Lemma 19, each iteration of the algorithm runs in $O(\log\log\Delta)$ rounds. Since there are $O(\sqrt{\log\Delta})$ iterations for $f = 2^{\sqrt{\log\Delta}}$, the total number of rounds is $O(\sqrt{\log\Delta} \cdot \log\log\Delta)$. From Lemma 21, we see that the sparsified graph given by M together with vertices still in V has degree at most $2^{O(\sqrt{\log\Delta})}$. Therefore, the MIS computation at the end of the algorithm takes $O(\sqrt{\log\Delta} + \log\log^* n)$ by using the deterministic MIS algorithm from Lemma 27 of [20] that runs in $O(\log\Delta' + \log\log^* n)$ on a Δ'-maximum degree graph, provided that the allowed global space is $O(n^{1+\delta} + m)$. Otherwise, we use the variation given in [23] that runs in $O(\sqrt{\log\Delta} \cdot \log\log n)$ and uses linear global space. ◀

Lastly, we need to show how to achieve a poly(Δ) coloring of G^2 to fulfill the assumption made in Lemma 17. Due to space constraints, it is deferred to Appendix B.1.

References

1 Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7(4):567–583, 1986. doi:10.1016/0196-6774(86)90019-2.

2 Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, 2013. URL: https://api.semanticscholar.org/CorpusID:316401.

3 Alkida Balliu, Sebastian Brandt, Juho Hirvonen, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. Lower bounds for maximal matchings and maximal independent sets. *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 481–497, 2019. URL: https://api.semanticscholar.org/CorpusID:57721262.

4 Alkida Balliu, Sebastian Brandt, Fabian Kuhn, and Dennis Olivetti. Distributed delta-coloring plays hide-and-seek. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, pages 464–477, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519935.3520027.

5 Alkida Balliu, Sebastian Brandt, and Dennis Olivetti. Distributed lower bounds for ruling sets. *SIAM Journal on Computing*, pages 70–115, 2022. URL: https://epubs.siam.org/doi/10.1137/20M1381770, doi:10.1137/20M1381770.

6 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *J. ACM*, 63(3), June 2016. doi:10.1145/2903137.

7 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, 2013. URL: https://api.semanticscholar.org/CorpusID:11086753.

8 M. Bellare and J. Rompel. Randomness-efficient oblivious sampling. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 276–287, 1994. doi:10.1109/SFCS.1994.365687.

9 Andrew Berns, James Hegeman, and Sriram V. Pemmaraju. Super-fast distributed algorithms for metric facility location. *ArXiv*, abs/1308.2473, 2012. URL: https://api.semanticscholar.org/CorpusID:124685, arXiv:1308.2473.

10 Tushar Bisht, Kishore Kothapalli, and Sriram V. Pemmaraju. Brief announcement: Super-fast t-ruling sets. *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, 2014. URL: https://api.semanticscholar.org/CorpusID:12210091.

11 Mélanie Cambus, Fabian Kuhn, Shreyas Pai, and Jara Uitto. Time and Space Optimal Massively Parallel Algorithm for the 2-Ruling Set Problem. In Rotem Oshman, editor, *37th International Symposium on Distributed Computing (DISC 2023)*, volume 281 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:12, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2023.11.

12 Keren Censor-Hillel, Merav Parter, and Gregory Schwartzman. Derandomizing local distributed algorithms under bandwidth restrictions. *Distributed Computing*, 33(3):349–366, June 2020. doi:10.1007/s00446-020-00376-1.

13 Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The Complexity of (Delta+1) Coloring in Congested Clique, Massively Parallel Computation, and Centralized Local Computation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 471–480, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331607.

14 Benny Chor and Oded Goldreich. On the power of two-point based sampling. *Journal of Complexity*, 5(1):96–106, 1989. doi:10.1016/0885-064X(89)90015-0.

15 Sam Coy and Artur Czumaj. Deterministic massively parallel connectivity. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, pages 162–175, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519935.3520055.

16 Artur Czumaj, Peter Davies, and Merav Parter. Simple, deterministic, constant-round coloring in the congested clique. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 309–318, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405751.

17 Artur Czumaj, Peter Davies, and Merav Parter. Component stability in low-space massively parallel computation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 481–491, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467903.

18 Artur Czumaj, Peter Davies, and Merav Parter. Graph sparsification for derandomizing massively parallel computation with low space. *ACM Trans. Algorithms*, 17(2), May 2021. doi:10.1145/3451992.

19 Artur Czumaj, Peter Davies, and Merav Parter. Improved Deterministic (Delta+1) Coloring in Low-Space MPC. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 469–479, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467937.

20 Artur Czumaj, Peter Davies-Peck, and Merav Parter. Component stability in low-space massively parallel computation. *Distributed Computing*, 37(1):35–64, March 2024. doi:10.1007/s00446-024-00461-9.

21 Guy Even, Oded Goldreich, Michael Luby, Noam Nisan, and Boban Veličković. Efficient approximation of product distributions. *Random Structures & Algorithms*, 13(1):1–16, 1998. doi:10.1002/(SICI)1098-2418(199808)13:1<1::AID-RSA1>3.0.CO;2-W.

22 Manuela Fischer, Jeff Giliberti, and Christoph Grunau. Improved Deterministic Connectivity in Massively Parallel Computation. In Christian Scheideler, editor, *36th International Symposium on Distributed Computing (DISC 2022)*, volume 246 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:17, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2022.22.

23 Manuela Fischer, Jeff Giliberti, and Christoph Grunau. Deterministic massively parallel symmetry breaking for sparse graphs. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '23, pages 89–100, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3558481.3591081.

24 Beat Gfeller and Elias Vicari. A randomized distributed algorithm for the maximal independent set problem in growth-bounded graphs. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2007. URL: https://api.semanticscholar.org/CorpusID:13473182.

25 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the 2016 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 270–277, 2016. doi:10.1137/1.9781611974331.ch20.

26 Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrović, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for mis, matching, and vertex cover. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 129–138, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3212734.3212743.

27 Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional hardness results for massively parallel computation from distributed lower bounds. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1650–1663, 2019. doi:10.1109/FOCS.2019.00097.

28 Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1636–1653, 2019. doi:10.1137/1.9781611975482.99.

29 Michael T. Goodrich. Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2):416–432, 1999. doi:10.1137/S0097539795294141.

30 Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In Takao Asano, Shin-ichi Nakano, Yoshio Okamoto, and Osamu Watanabe, editors, *Algorithms and Computation*, pages 374–383, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-25591-5_39.

31 James Hegeman, Sriram V. Pemmaraju, and Vivek Sardeshmukh. Near-constant-time distributed algorithms on a congested clique. In *International Symposium on Distributed Computing*, 2014. URL: https://api.semanticscholar.org/CorpusID:277941.

32 Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the 2010 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010. doi:10.1137/1.9781611973075.76.

33 Kishore Kothapalli, Shreyas Pai, and Sriram V. Pemmaraju. Sample-And-Gather: Fast Ruling Set Algorithms in the Low-Memory MPC Model. In Nitin Saxena and Sunil Simon, editors, *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2020)*, volume 182 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSTTCS.2020.28.

34 Kishore Kothapalli and Sriram Pemmaraju. Distributed graph coloring in a few rounds. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 31–40, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993806.1993812.

35 Kishore Kothapalli and Sriram V. Pemmaraju. Super-fast 3-ruling sets. In *Foundations of Software Technology and Theoretical Computer Science*, 2012. URL: https://api.semanticscholar.org/CorpusID:16038481.

36 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *J. ACM*, 63(2), 2016. doi:10.1145/2742012.

37 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

38 Michael Luby. Removing randomness in parallel computation without a processor penalty. *Journal of Computer and System Sciences*, 47(2):250–286, 1993. doi:10.1016/0022-0000(93)90033-S.

39 Yannic Maus, Saku Peltonen, and Jara Uitto. Distributed symmetry breaking on power graphs via sparsification. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, PODC '23, pages 157–167, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3583668.3594579.

40 Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms.* Cambridge university press, 1995. doi:10.1017/CBO9780511814075.

41 Rajeev Motwani, Joseph (Seffi) Naor, and Moni Naor. The probabilistic method yields deterministic parallel algorithms. *Journal of Computer and System Sciences*, 49(3):478–516, 1994. 30th IEEE Conference on Foundations of Computer Science. doi:10.1016/S0022-0000(05)80069-8.

42 Krzysztof Nowicki. A deterministic algorithm for the mst problem in constant rounds of congested clique. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 1154–1165, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3406325.3451136.

43 Shreyas Pai and Sriram V. Pemmaraju. Brief announcement: Deterministic massively parallel algorithms for ruling sets. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, pages 366–368, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519270.3538472.

44 Prabhakar Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. *Journal of Computer and System Sciences*, 37(2):130–143, 1988. doi:10.1016/0022-0000(88)90003-7.

45 Johannes Schneider, Michael Elkin, and Roger Wattenhofer. Symmetry breaking depending on the chromatic number or the neighborhood growth. *Theor. Comput. Sci.*, 509:40–50, 2013. doi:10.1016/J.TCS.2012.09.004.

A Missing Proofs for Linear MPC Result

Proof of Lemma 12. We analyze one step of (a variation of) Luby's algorithm that builds an independent set $\mathcal{I}$ on the set of sampled bad vertices $\bigcup_d B_d \cap V_{\text{samp}}$. We will fix a seed specifying a hash function from a pairwise independent family $\mathcal{H}$. Let $v \in (\bigcup_d B_d \cap V_{\text{samp}})$. An hash function h maps node v to a value $z_v \in [n^3]$. Then, v joins the independent set $\mathcal{I}$ iff $z_v < z_w$ for all $w \sim v$ *and* $z_v < \frac{n^3}{d^{3\varepsilon}}$, where $w \in N(v) \cap (\bigcup_d B_d \cap V_{\text{samp}})$.

By Lemma 10, each node $u \in \hat{B}_d$ has at least $d^{0.1}$ nodes from S_u that are sampled, each of which has at most $d^{2\varepsilon}$ sampled neighbors. For the purpose of the analysis, let the set A_u include exactly $d^{0.1} = d^{4\varepsilon}$ of such nodes and let $\{X_v\}_{v\in A_u}$ be the random variables denoting the event that v joins $\mathcal{I}$. We denote $X = \sum_{v\in A_u} X_v$ as their sum. For any v, we have

$$\frac{1}{d^{3\varepsilon}} - \frac{1}{n^3} \le \Pr\left[z_v < \frac{n^3}{d^{3\varepsilon}}\right] \le \frac{1}{d^{3\varepsilon}}.$$

By pairwise independence,

$$\Pr[X_v = 1] \ge \Pr\left[z_v < \frac{n^3}{d^{3\varepsilon}}\right] - \sum_{v' \in N(v)\cap S(B)} \Pr\left[z_{v'} \le z_v < \frac{n^3}{d^{3\varepsilon}}\right] \ge \frac{1}{d^{3\varepsilon}} - \frac{1}{n^3} - \frac{d^{2\varepsilon}}{d^{6\varepsilon}} \ge \frac{1}{3d^{3\varepsilon}}.$$

It follows that $\mathbb{E}[X] = \sum_{v\in A_u} \Pr[X_v = 1] \ge \frac{d^{\varepsilon}}{3}$. Our goal is now to bound $\Pr[X = 0]$. Observe that for any two vertices $v, v' \in A_u$, we have that

$$\mathbb{E}[X_v X_{v'}] \le \Pr\left[z_v < \frac{n^3}{d^{3\varepsilon}} \cap z_{v'} < \frac{n^3}{d^{3\varepsilon}}\right] \le d^{-6\varepsilon},$$

by pairwise independence. Thus, we get

$$\frac{\mathbb{V}\text{ar}[X]}{\mathbb{E}[X]^2} \le \frac{\sum_{v\in A_u} \mathbb{V}\text{ar}[X_v] + \sum_{v,v'\in A_u} \text{Cov}[X_v, X_{v'}]}{\mathbb{E}[X]^2}.$$

We know that

$$\sum_{v \in A_u} \mathbb{V}\mathrm{ar}[X_v] \le d^{4\varepsilon} \cdot \Pr[X_v = 1](1 - \Pr[X_v = 1]) \le d^{4\varepsilon} \cdot \frac{1}{3d^{3\varepsilon}} = \frac{d^{\varepsilon}}{3},$$

$$\sum_{v,v' \in A_u} \mathrm{Cov}[X_v, X_{v'}] \le d^{8\varepsilon}(\mathbb{E}[X_v X_{v'}] - \mathbb{E}[X_v]\mathbb{E}[X_{v'}]) \le d^{8\varepsilon}(d^{-6\varepsilon} - 1/9d^{6\varepsilon}) \le d^{2\varepsilon}.$$

Therefore,

$$\mathbb{V}\mathrm{ar}[X] \le \frac{d^{\varepsilon}}{3} + d^{2\varepsilon} \le \frac{4d^{\varepsilon}}{3}, \text{ and } \frac{\mathbb{V}\mathrm{ar}[X]}{\mathbb{E}[X]^2} \le \frac{\frac{4d^{\varepsilon}}{3}}{\left(\frac{d^{\varepsilon}}{3}\right)^2} = \frac{4d^{\varepsilon}}{3} \cdot \frac{9}{d^{2\varepsilon}} = \frac{36}{d^{\varepsilon}}.$$

Applying Chebyshev's inequality, we have

$$\Pr[X = 0] \le \Pr\left[|X - \mathbb{E}[X]| \ge \mathbb{E}[X]\right] \le \frac{\mathbb{V}\mathrm{ar}[X]}{\mathbb{E}[X]^2} \le \frac{45}{d^{\varepsilon}}.$$ ◀

B Missing Proofs for Low-Memory MPC Result

Proof of Lemma 20. Consider a high-degree vertex $u \in U$ at the start of the i-th iteration. By Lemma 19, each node in U is incident to a node that joins the set M by the end of this iteration. Since all vertices incident to M are removed from V, the lemma follows. ◀

Proof of Lemma 21. First, consider a vertex v that joins the set M at some iteration j. Observe that no neighbor of v in G had joined M earlier, otherwise, u would have been removed. By Lemma 19, all vertices that join M at iteration j have induced degree at most $2^{O(\log f)}$. Then, the neighbors of M are removed from V and, thus, cannot join M anymore. This proves that vertices in M have degree at most $2^{O(\log f)}$. Second, consider a vertex w that at the end of the $\lfloor \log f \rfloor$-th iteration is still in V. This means that w does not neighbor M and that, by Lemma 20, w has degree at most $2^{O(\log f)}$, finishing the claim. ◀

B.1 Coloring of G^2

Here, we discuss how to compute a poly(Δ) coloring of G^2 to fulfill the assumption made in Lemma 17.

Whenever $\Delta = n^{\Omega(1)}$, the initial assignment of IDs to vertices, typically from 1 to n, effectively serves as a poly(Δ) coloring of G^2. In the case where $\Delta \le n^{\delta}$ for constant $\delta < \alpha/2$, we ensure $\Delta^2 \ll n^{\alpha}$. This implies that the 2-hop neighborhood of every node can be stored within the local memory of a single machine. Storing the 2-hop neighbors on a single machine permits the use of Linial's coloring reduction technique [37], which achieves a $O(\Delta^6)$ coloring in $O(1)$ rounds. However, this approach necessitates of a global space usage of $O(n^{1+2\delta})$, potentially exceeding $O(n + m)$. To improve the global space usage, after three runs of Lemma 17, the degree of each vertex which has not been removed is at most $\Delta^{0.22}$. Since each sampled vertex is incident to a high-degree vertex of initial degree at least $O(\Delta/f)$, we can charge high-degree vertices $O(\Delta^{0.66}) \ll \Delta/f$ space consumption. This reduction allows us to gather the 2-hop neighbors of all active nodes onto single machines without breaching the global space limit. A further optimization involves substituting the first three runs of Lemma 17 with a weaker version, detailed below, addressing all but at most $\frac{n}{\Delta^{0.01}}$ vertices. The proof follows from that of Lemma 17.

▶ **Lemma 22.** *Let $G = (V, E)$ be a graph with an upper bound Δ on the maximum degree. There is a sublinear MPC algorithm that computes in $O(1)$ rounds a subset $V' \subseteq V$ ensuring that, for all but at most $\frac{n}{\Delta^{0.01}}$ vertices $v \in V$ with $\deg_G(v) \geq \log(n) \cdot \Delta^{0.6}$, it holds that $|N_G(v) \cap V'| \in \left[\frac{1}{3\sqrt{\Delta}}|N_G(v)|, \frac{1}{\sqrt{\Delta}}|N_G(v)|\right]$.*

Applying Lemma 22 initially and excluding up to $\frac{n}{\Delta^{\Omega(1)}}$ vertices not meeting our criteria allows for the execution of $O(\log\log\Delta)$ iterations for the well-behaved vertices. The excluded vertices are subsequently addressed by repeating the same process. After $O(1)$ iterations, the remaining vertex count drops to $O(\frac{n}{\Delta^2})$, fitting the global space needed to store their 2-hop neighborhoods within $O(n)$. Consequently, after $O(\log\log\Delta)$ rounds, all vertices are processed without affecting the asymptotic total number of rounds.

Granular Synchrony

Neil Giridharan[1]
Unversity of California, Berkeley, CA, USA

Ittai Abraham[1]
Intel Labs, Petah Tikva, Israel

Natacha Crooks
Unversity of California, Berkeley, CA, USA

Kartik Nayak
Duke University, Durham, NC, USA

Ling Ren[1]
University of Illinois Urbana-Champaign, IL, USA

Abstract

Today's mainstream network timing models for distributed computing are synchrony, partial synchrony, and asynchrony. These models are coarse-grained and often make either too strong or too weak assumptions about the network. This paper introduces a new timing model called granular synchrony that models the network as a mixture of synchronous, partially synchronous, and asynchronous communication links. The new model is not only theoretically interesting but also more representative of real-world networks. It also serves as a unifying framework where current mainstream models are its special cases. We present necessary and sufficient conditions for solving crash and Byzantine fault-tolerant consensus in granular synchrony. Interestingly, consensus among n parties can be achieved against $f \geq n/2$ crash faults or $f \geq n/3$ Byzantine faults without resorting to full synchrony.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Timing model, synchrony, asynchrony, consensus, blockchain, fault tolerance

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.30

Funding *Ling Ren*[1]: National Science Foundation award #2143058.

1 Introduction

A fundamental aspect of any distributed computation is the *timing model.* There are three mainstream timing models: synchrony, asynchrony, and partial synchrony. Under synchrony, messages arrive before a known upper bound Δ. Under asynchrony, messages arrive in any finite amount of time. With partial synchrony [16], there is an unknown but finite Global Stabilization Time (GST), and the network is asynchronous before GST and synchronous afterwards.

The synchrony model is arguably a rosy reality: even a single message that takes longer than Δ to arrive is a violation of the synchrony model (forcing us to consider either the sender or recipient to be faulty). On the other hand, the asynchrony model is extremely pessimistic, making it challenging, or even impossible, to design protocols in it. The most well-known example may be the FLP impossibility [18], which states that any consensus protocol that can tolerate even a single crash fault in asynchrony must have an infinite execution. This implies that deterministic consensus in asynchrony is impossible. The partial

[1] This work was started while authors were at VMware Research.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 30; pp. 30:1–30:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

synchrony model tries to balance asynchrony and synchrony and has been the most widely adopted in practice so far. But it is close to asynchrony in essence and shares the same fault tolerance bounds as (randomized) asynchronous protocols.

This paper argues that the current characterization of network timings is too coarse-grained. We recognize the variability and heterogeneity of modern networks and propose that they should be modeled in a *granular manner* via a graph consisting of a mixture of synchronous, partially synchronous, and asynchronous links. We call the new model *granular synchrony.*

Our new model is more than yet another theoretical construct. It is rooted in and motivated by our understanding and characterizations of modern distributed systems and networks. Modern distributed systems increasingly span datacenters, be it for disaster recovery or fault isolation [32, 6, 28]. Within datacenters, networks are mostly synchronous [35]. Spikes in message delays do occur [3], but such spikes are rare and almost never happen to the entire datacenter [21]. Across datacenters and over the Internet, networks are mostly well-behaved but are susceptible to significant fluctuations [22] and adversarial attacks [14].

The granular synchrony timing model can serve as a unifying framework for network timing models. Synchrony, partial synchrony, and asynchrony are all extreme cases of it. Outside these extreme cases, the granular synchrony model is a natural intermediate between synchrony and partial synchrony (or asynchrony) and gives rise to new results that can be construed as an intermediate between fundamental results in distributed computing.

For concreteness, we focus on the problem of fault-tolerant consensus [27] in this paper. It is well-known that under synchrony, the agreement variant of consensus can be solved in the presence of $f < n$ crash faults or $f < n/2$ Byzantine faults (assuming digital signatures). With partial synchrony, fewer faults can be tolerated: $f < n/2$ crash faults or $f < n/3$ Byzantine faults [16]. Asynchrony has the same fault thresholds and further requires the use of randomization [18].

We derive necessary and sufficient conditions for solving crash fault-tolerant (CFT) and Byzantine fault-tolerant (BFT) consensus in granular synchrony. A key benefit and interesting implication of the granular synchrony model is that we do *not* have to assume full synchrony to tolerate $f \geq n/2$ crash faults or $f \geq n/3$ Byzantine faults. Instead, consensus can be reached if and only if the underlying communication graph satisfies certain conditions.

We remark that all our protocols are *graph-agnostic*, meaning they do not need to know the synchronicity property of any link. As a result, our protocols can work in the following alternative formulation of the granular synchrony model. The consensus algorithm is parameterized by n and f. Initially, all communication links are synchronous. The adversary has the power to corrupt f nodes and alter some links to be partially synchronous or asynchronous but must not violate the necessary condition for the given n and f. On the other hand, most of our impossibility proofs rule out algorithms that know the graph and are tailored for the graph. This strengthens both our protocols and our impossibility results.

We will consider two variants of the granular synchrony model. The first variant only has synchronous and partially synchronous links (no asynchronous links), and we refer to it as *granular partial synchrony.* CFT consensus in granular partial synchrony can be solved if and only if any quorum of $n - f$ nodes collectively can communicate synchronously with at least $f + 1$ nodes despite faulty nodes. BFT consensus in granular partial synchrony can be solved if and only if any set of $n - 2f$ correct nodes can communicate synchronously with at least $f + 1$ correct nodes despite faulty nodes.

The second variant further allows asynchronous links, and we refer to it as *granular asynchrony.* For CFT consensus to be solved deterministically in granular asynchrony, it is additionally required that after removing all asynchronous edges and all crashed nodes, less than $n - f$ nodes are outside the largest connected component of the remaining graph. For undirected graphs, this condition is weaker than the correct $\diamond f$-source condition in [4] (see §B) and establishes the minimum synchrony condition needed to circumvent the FLP impossibility [18]. For BFT consensus to be solved deterministically in granular asynchrony by a *graph-agnostic* algorithm, it is additionally required that there is a correct node with partially synchronous paths to at least f other correct nodes. The necessary and sufficient condition for algorithms that know the graph is still open.

2 Model and Definitions

We assume communication links are bi-directional. In granular partial synchrony, each link can be either synchronous or partially synchronous. In granular asynchrony, each link can be synchronous, partially synchronous, or asynchronous. A synchronous link delivers each message sent on the link within a known upper bound Δ. A partially synchronous link respects the Δ message delivery bound after GST. An asynchronous link has no delay bound and just has to deliver each message eventually. We assume all communication links are reliable and FIFO (first-in-first-out), and deliver each transmitted message exactly once.

Beyond this, the model is the same as traditional consensus literature. There are n nodes in total. The adversary can corrupt up to f nodes and can do so at any time during the protocol execution (i.e., the adversary is adaptive). In the CFT case, faulty nodes can fail by crashing only. In the BFT case, faulty nodes can behave arbitrarily and can be coordinated by the adversary. For BFT, we further assume the existence of digital signatures and public-key infrastructure (PKI) and that faulty nodes cannot break cryptographic primitives. A message is only considered valid by correct nodes if its accompanying signature is verified (we omit writing these signature operations in the protocols).

Our protocols do not require any form of clock synchronization among nodes, and instead just require bounded clock skews. To elaborate, certain steps of our protocols require nodes to wait for some amount of time (e.g., 4Δ). For simplicity, our protocol description assumes each node will wait for precisely that amount of time. But it is not hard to see that our protocols still work if each node waits for a time that falls in a known bounded range (e.g., between 4Δ and 5Δ), which is easy to achieve with bounded clock skews.

It is convenient to describe the network as an undirected graph $G = (V, E)$. Each vertex represents a node, and each edge represents a communication link. We use vertex and node interchangeably, and edge and link interchangeably. Our protocols are graph agnostic: they do not assume knowledge of the graph.

▶ **Definition 1** (Synchronous path). *Node a has a synchronous path to node b, written as $a \to b$, if there exist a sequence of synchronous edges $(a, i_1), (i_1, i_2), , \dots, (i_k, b)$ where every intermediate node i_j is correct.*

Note that in the above definition, only intermediate nodes need to be correct. Therefore, every node, even a faulty one, has a synchronous path to itself, i.e., $a \to a, \forall a \in V$. We generalize the notion of synchronous paths from two nodes to two sets of nodes A and B.

▶ **Definition 2.** *$A \to B$ if $\forall b \in B, \exists a \in A$ such that $a \to b$.*

▶ **Definition 3** (Path length, distance and diameter). *The length of a path is the number of edges in it. If $a \to b$, the synchronous distance between these two nodes is the length of the shortest synchronous path between them. The synchronous diameter of a graph G is*

$$d(G) := \max_{F,a,b \text{ s.t. } |F| \le f,\ a \to b} d(a, b).$$

Partially synchronous path, path length, distance, and diameter $d'(G)$ are similarly defined. Note that a partially synchronous path can contain synchronous edges.

The (partially) synchronous distance is only defined for a pair of nodes that have a (partially) synchronous path between them. We also remark that for the Byzantine case, distance is only defined for a pair of correct nodes. The max in the diameter definition is taken over all pairs with the corresponding distance defined. The two diameters capture the worst-case round-trip delays among nodes connected by synchronous and partially synchronous paths, respectively. If $d(G)$ or $d'(G)$ is known, they can be directly used in our protocols; otherwise, $|V| - 1$ is a trivial upper bound. We will simply write d and d' when there is no ambiguity.

▶ **Definition 4** (Consensus). *In a consensus protocol, every node has an initial input value and must decide a value that satisfies the following properties.*

- *Agreement: No two correct nodes decide different values.*[2]
- *Termination: Every correct node eventually decides.*
- *Validity: If all nodes have the same input value, then that is the decision value.*

3 CFT Consensus in Granular Partial Synchrony

▶ **Theorem 5.** *Under granular partial synchrony, CFT consensus on a graph $G = (V, E)$ is solvable if and only if, regardless of which up to f nodes are faulty, $\forall A \subseteq V$ with $|A| \geq n - f$, $\exists B \subseteq V$ with $|B| \geq f + 1$ such that $A \rightarrow B$.*

In words, the condition is that any set A of size at least $n - f$ has a potentially larger set B of size at least $f + 1$, such that for any node $b \in B$ there exits $a \in A$ and a synchronous path from a to b. Intuitively, if a message arrives at all of A, then it will arrive at all of B after some delay.

It is worth noting that classic crash fault tolerance bounds are special cases of our theorem. For example, when all links are synchronous, any node has synchronous paths to all n nodes. Thus, synchronous CFT consensus can be solved for any $n \geq f + 1$. At the other extreme, $n = 2f + 1$ is the smallest value of n for which the condition in Theorem 5 trivially holds even when all edges are partially synchronous (see necessity proof). The more interesting part of our theorem is of course when we have a mix of synchronous and partially synchronous edges. Figure 1 gives examples of these intermediate cases where CFT consensus is solvable with $f + 1 < n \leq 2f$.

3.1 Necessity

We first prove the "only if" part of Theorem 5. The proof is similar to the DLS proof in partial synchrony [16]. To ensure agreement, we must ensure that nodes cannot be partitioned into two disjoint groups with no synchronous inter-group links. The condition in Theorem 5 ensures exactly that.

Proof. For $n \geq 2f + 1$, the "only if" part of the theorem is vacuous because the condition trivially holds: $n - f \geq f + 1$, and every node has a synchronous path to itself.

For $n \leq 2f$, we prove by contradiction. Suppose there is an algorithm that solves consensus on a graph G that does not satisfy the condition in the theorem. Then, there exists a set F of up to f nodes such that, if nodes in F crash, there exists a set A of at least

[2] For CFT consensus, we actually achieve the stronger property of uniform agreement, which states that no two nodes (even faulty ones) decide differently.

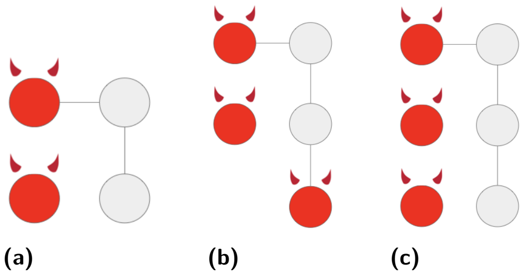


Figure 1 Only synchronous links are shown in the figure for brevity. Faulty nodes are denoted in red with horns, and the correct nodes are denoted in gray. The figure shows the necessary and sufficient condition in theorem 5 being satisfied for **(a)** $n = 4$, $f = 2$, **(b)** $n = 5$, $f = 3$, and **(c)** $n = 6$, $f = 3$.

$n - f$ nodes, which collectively have synchronous paths to at most f nodes. Let B be the set of these f nodes excluding A. Let C be the remaining nodes, i.e., $C = [n] \setminus \{A \cup B\}$. Note that $\{A, B, C\}$ is a three-way disjoint partition of the n nodes. Also note that $|A \cup B| \leq f$ and $|B \cup C| = n - |A| \leq f$. Next, we consider three executions.

In execution 1, all nodes have input v_1 and nodes in $B \cup C$ crash at the beginning. Since $|B \cup C| \leq f$, A eventually decides v_1 in time t_1 due to validity. In execution 2, all nodes have input $v_2 \neq v_1$ and nodes in $A \cup B$ crash at the beginning. Since $|A \cup B| \leq f$, C eventually decides v_2 in time t_2 due to validity.

In execution 3, nodes in A have input v_1, nodes in C have input v_2, nodes in B crash at the beginning, and $GST > \max(t_1, t_2)$. Note that crashing B (instead of F) does not change the fact that A has synchronous paths to $A \cup B$ only. This is because, with B crashed, nodes in $F \setminus B$ do not have synchronous paths to A themselves (otherwise, they would have synchronous paths to A with F crashed). Thus, synchronous paths from A to C cannot go through $F \setminus B$. Because there are no synchronous edges between A and C, the adversary can delay the delivery of all messages between A and C until after GST. Thus, A cannot distinguish execution 3 from execution 1 and C cannot distinguish execution 3 from execution 2. Then, A decides v_1 and C decides v_2, violating agreement. ◀

3.2 Protocol

Next, we present a new CFT consensus protocol assuming the condition in theorem 5 holds. This establishes the sufficiency of the condition.

Overview. A natural starting point is a standard quorum-based partially synchronous CFT consensus protocol. Such protocols require $n > 2f$ to ensure any two quorums of size $n - f$ intersect. When $n \leq 2f$, two quorums of $n - f$ may not intersect. But when the condition in theorem 5 holds, a quorum of $n - f$ nodes can hear from $f + 1$ nodes of any critical information in bounded time. This effectively promotes a quorum of size $n - f$ to $f + 1$ and ensures safety as a quorum of size $f + 1$ always intersects a quorum of size $n - f$.

Similar to other leader-based partially synchronous consensus protocols, our protocol operates in a series of views, where each view has a leader. The leader of view v is denoted as L_v. Leaders can be elected using a simple round-robin order. If a view after GST has a correct leader, nodes will commit that leader's proposal and terminate. There is a view change procedure to replace a leader who is not making progress. We focus on a single-shot consensus here, but the protocol can be easily adapted to the multi-shot setting.

Algorithm 1 CFT consensus protocol in granular partial synchrony for node i.

```
$v_i \leftarrow 0$                                              ▷ Initialize local view number
$lock \leftarrow (0, input_i)$                                  ▷ Initially lock on the input value
**enter** view 1

**upon** entering view $v$ **do**
    $v_i \leftarrow v$
    **start** $view\_timer \leftarrow timer(4\Delta)$           ▷ Timer for changing view
    **send** $\langle$STATUS, $v, lock\rangle$ to $L_v$

**upon** receiving $n - f$ $\langle$STATUS, $v_i, -\rangle$ and $i = L_{v_i}$ **do**
    $val \leftarrow$ value from the highest lock (by view) received
    **send** $\langle$PROPOSE, $v_i, val\rangle$ to all          ▷ Leader proposal

**upon** receiving $\langle$PROPOSE, $v_i, val\rangle$ **do**
    $lock \leftarrow (v_i, val)$
    **send** $\langle$VOTE, $v_i, val\rangle$ to all

**upon** receiving $n - f$ $\langle$VOTE, $v_i, val\rangle$ or $\langle$COMMIT, $val\rangle$ **do**
    **send** $\langle$COMMIT, $val\rangle$ to all
    **commit** $val$ and **terminate**

**upon** $view\_timer$ expiring **do**
    **send** $\langle$NEWVIEW, $v_i + 1\rangle$ to all

**upon** receiving $\langle$NEWVIEW, $v\rangle$ where $v > v_i$ **do**
    **echo** $\langle$NEWVIEW, $v\rangle$ and to all
    **send** $\langle$LOCKED, $lock\rangle$ to all
    **stop** accepting PROPOSE messages in views up to $v - 1$
    **wait** $2d\Delta$ time
    **enter** view $v$

**upon** receiving $\langle$LOCKED, $lock'\rangle$ **do**
    $lock \leftarrow$ higher lock (by view) between $lock$ and $lock'$
    **echo** $\langle$LOCKED, $lock'\rangle$ to all
```

Locks. A $lock := (view, value)$ consists of a view and value. Initially, each node locks on its input value with view number 0. When a node receives a proposal from the leader of the current view, it updates its lock to the current view and the proposed value. Locks are ranked by view numbers. Note that except for the initial view 0, there cannot be two locks with the same view number but different values, since only one value is proposed per view. Locks from view 0 can be ranked arbitrarily.

We describe the protocol next.

Status step. Each view begins with every node sending a STATUS message to the leader of the current view. A node also starts a timer for the view.

Leader proposal step. When L_v is in view v and receives $n - f$ $\langle$STATUS, $v, -\rangle$ messages, it proposes the highest locked value among those. Note that L_v only sends one PROPOSE message in a view. When a node is in view v and receives a $\langle$PROPOSE, $v, val\rangle$ message, it updates its lock to (v, val) and sends a $\langle$VOTE, $v, val\rangle$ message to all nodes.

Commit step. When a node receives a quorum of $n - f$ $\langle$Vote, $v, val\rangle$ messages or a single $\langle$Commit, $val\rangle$ message, it commits val, sends a $\langle$Commit, $val\rangle$ message to all nodes, and terminates.

View change step. When a node times out in a view v without committing a value, it sends $\langle$NewView, $v + 1\rangle$ to all nodes, asking them to move to the next view. Upon receiving $\langle$NewView, $v\rangle$ for a higher view v, a node echoes $\langle$NewView, $v\rangle$ and its own lock to all nodes, waits for $2d\Delta$ time, and then enters view v. During this waiting period, the node will not send Vote for its current view but will listen for Locked messages to update its lock and also echo locks. The $2d\Delta$ time accounts for the worst-case round-trip delay to send a NewView message and receive the Locked message.

3.3 Analysis

▶ **Lemma 6.** *If some node commits val in view v, then any $\langle$Propose, $v', val'\rangle$ message in view $v' \geq v$ must have $val' = val$.*

Proof. We prove this lemma by induction on view v'. The base case of $v' = v$ is straightforward since each leader proposes only one value, so $val' = val$.

For the inductive step, suppose the lemma holds up to view $v' - 1$, and we consider view v'. Suppose for the sake of contradiction that some node commits val in view v, and there is a $\langle$Propose, $v', val'\rangle$ message from $L_{v'}$ for $val' \neq val$. $L_{v'}$ must have received $\langle$Status, $v', -\rangle$ messages from a set P of $n - f$ nodes. By the condition in theorem 5, $P \to Q$, where Q is a set of $f + 1$ nodes. Since a node committed val in view v, there must exist a set R of $n - f$ nodes that sent $\langle$Vote, $v, val\rangle$ messages and updated $lock := (v, val)$ in view v. Sets Q and R intersect in at least one node. Let this node be q.

Since the graph is undirected, there must exist a node $p \in P$ such that $q \to p$. By the induction hypothesis, Propose messages from view v to $v' - 1$ must be for val. Since a node only updates its lock monotonically based on view numbers, node q must have a lock with view $\geq v$ for val. Let t_p be the time node p echoed $\langle$NewView, $v'\rangle$. By time $t_p + d\Delta$, node q receives $\langle$NewView, $v'\rangle$. Upon receiving $\langle$NewView, $v'\rangle$, node q sends a $\langle$Locked, $lock\rangle$ message to all nodes. This lock is received by node p by time $t_p + 2d\Delta$. Node p updates its lock to view $\geq v$ for val before entering view v'. Thus, L_v receives at least one Status message for val with view $\geq v$ and propose val, a contradiction. ◀

▶ **Theorem 7** (Agreement). *No two nodes commit different values.*

Proof. Let v be the smallest view in which a node commits some value, say val. Since only val can be proposed in view v and all subsequent views by lemma 6, no node can commit a different value. ◀

▶ **Theorem 8** (Termination). *All correct nodes eventually decide.*

Proof. With round-robin leader election, correct nodes are elected leaders infinitely often. Thus, there must be a view v, after $GST + 2d\Delta$, whose leader is correct. We next prove that all nodes will decide and terminate in view v (if they don't decide earlier).

Let t ($t \geq GST + 2d\Delta$) be the first time some correct node enters view v. This correct node sends $\langle$NewView, $v\rangle$ to all nodes at $t - 2d\Delta \geq GST$. All correct nodes receive $\langle$NewView, $v\rangle$ by time $t - 2d\Delta + \Delta$, wait $2d\Delta$ themselves, and enter view v by time $t + \Delta$. Upon entering view v, they send $\langle$Status, $v, -\rangle$ messages to L_v. L_v receives $n - f$ $\langle$Status, $v, -\rangle$ messages by time $t + 2\Delta$, and sends a $\langle$Propose, $v, -\rangle$ message to all nodes. All correct nodes receive the $\langle \textsc{Propose}, v, - \rangle$ message and send $\langle \textsc{Vote}, v, - \rangle$ messages by time $t + 3\Delta$. All correct nodes receive $n - f$ $\langle \textsc{Vote}, v, - \rangle$ messages and commit by time $t + 4\Delta$. Since a node's view timer is 4Δ, all correct nodes commit and terminate in view v. ◀

▶ **Theorem 9** (Validity). *If all nodes have the same input val, then all correct nodes eventually decide val.*

Proof. If all nodes have the same input val, all nodes set $lock \leftarrow (0, val)$. Following a similar proof as in lemma 6, no other value can be proposed in all subsequent views. Validity follows from termination. ◀

4 CFT Consensus in Granular Asynchrony

▶ **Theorem 10.** *Under granular asynchrony, CFT consensus on a graph $G = (V, E)$ can be solved deterministically if and only if, (i) the condition in theorem 5 holds and (ii) for all F with $|F| \leq f$, less than $n - f$ nodes are outside the largest connected component of $G' = (V - F, \diamond E)$ where $\diamond E$ is the set of synchronous and partially synchronous edges.*

In other words, condition (ii) says that if we remove all asynchronous edges and all faulty nodes from G and further remove the largest connected component in the remaining graph, then there are fewer than $n - f$ nodes left.

4.1 Necessity

Proof. Condition (i) is already proved to be necessary in theorem 5. We focus on condition (ii). Suppose for the sake of contradiction there exists a deterministic algorithm $\mathcal{A}$ that solves CFT consensus on a graph G that violates condition (ii). This means there exists a set F with $|F| \leq f$ such that removing the largest connected component from $G' = (V - F, \diamond E)$ (G with F and all asynchronous edges removed) leaves $\geq n - f$ nodes.

Suppose the graph G' has q connected components. Clearly, $q > 1$. Let C_i be i-th connected component in G'. We have $|F \cup C_i| \leq f$ for all i because even the largest connected component plus F has at most f nodes.

We construct an external system consisting of q nodes connected only by asynchronous links. We can convert $\mathcal{A}$ into a deterministic algorithm that solves consensus in this external system while tolerating one crash fault. To do so, let the i-th node in the external system, q_i, simulate the nodes in C_i in $\mathcal{A}$. If q_i has input v_i, then all nodes in C_i have input v_i in the simulation.

An execution in this external system with q_i crashing at time t faithfully simulates an execution of $\mathcal{A}$ with F crashing in the beginning and C_i crashing at time t. In particular, observe that two connected components in G' only have asynchronous edges between them once nodes in F crash. Since $|F \cup C_i| \leq f$ for all i, $\mathcal{A}$ solves consensus in the original system. Thus, the simulated algorithm solves consensus deterministically in the external system while tolerating one crash fault in asynchrony. This contradicts the FLP impossibility [18]. ◀

4.2 Protocol

Next, we adapt our previous CFT consensus protocol in algorithm 1 from granular partial synchrony to granular asynchrony, assuming the condition in theorem 10 holds. This establishes the sufficiency of the condition.

Our prior CFT consensus protocol still maintains safety under granular asynchrony, but liveness no longer holds because there is no time when all edges behave synchronously (asynchronous links do not have a GST assumption). As a result, correct leaders in our prior

Algorithm 2 CFT consensus protocol in granular asynchrony for node i.

```
$v_i \leftarrow 0$                                          ▷ Initialize local view number
$lock \leftarrow (0, input_i)$                              ▷ Initially lock on the input value
enter view 1

upon entering view $v$ do
    $v_i \leftarrow v$
    send ⟨Status, $v, lock$⟩ to all

upon receiving $n - f$ ⟨Status, $v_i, -$⟩ where $i \neq L_{v_i}$ do
    echo these $n - f$ ⟨Status, $v_i, -$⟩ to all
    start $proposal\_timer \leftarrow timer(3d'\Delta)$

upon receiving ⟨Propose, $v_i, val$⟩ do
    $lock \leftarrow (v_i, val)$
    echo ⟨Propose, $v_i, val$⟩ to all
    send ⟨Vote, $v_i, val$⟩ to all

upon $proposal\_timer$ expiring and no leader proposal received do
    send ⟨ViewChange, $v_i$⟩ to all

upon receiving $n - f$ ⟨ViewChange, $v$⟩ do
    send ⟨NewView, $v + 1$⟩ to all

Vote, Commit, Locked, NewView messages at all nodes and Status messages at
view leaders are processed the same way as in Algorithm 1
```

protocol may continuously time out. Luckily, condition (ii) in theorem 10 can be leveraged to guarantee that when the set F of crashed nodes stops growing, and a correct node in the largest connected component of $G' = (V - F, \diamond E)$ is elected leader after GST, this leader will not be replaced and will make progress. To do so, we first require $n - f$ nodes to initiate a view change. This way, because all nodes in F are crashed and fewer than $n - f$ nodes are outside the largest connected component of $G' = (V - F, \diamond E)$, we just need to make sure that no node in this largest connected component initiates a view change. This technique is similar to those used in view synchronizes [11, 10] to make sure correct nodes eventually overlap and remain in the same view to ensure termination.

We only describe the status and view change steps since the rest of the protocol remains the same as algorithm 1.

Status and propose step. Upon entering a new view v, a node sends a ⟨Status, $v, lock$⟩ message to **all** nodes. When a node receives at least $n - f$ ⟨Status, $v, -$⟩ messages, it forwards this set of Status messages to all nodes and starts a timer of $3d'\Delta$ duration. Upon receiving a proposal, a node forwards the proposal to all nodes, in addition to locking on and voting for the proposal. The same vote and commit steps from algorithm 1 follow.

View change. A node suspects the leader is faulty if it does not receive a ⟨Propose, $v, -$⟩ message before its timer expires. When this occurs, a node sends a ⟨ViewChange, v⟩ message to all nodes, indicating it wishes to quit view v. When a node receives $n - f$ ⟨ViewChange, v⟩ messages for the current view v, it sends a ⟨NewView, $v + 1$⟩ message to all nodes. Upon receiving a NewView message, a node carries out the same new view step from algorithm 1.

4.3 Analysis

The agreement and validity proofs are identical to the granular partial synchrony CFT case. We focus on termination.

▶ **Lemma 11.** *If no correct node ever terminates, then every correct node keeps entering higher views.*

Proof. Suppose for the sake of contradiction, there exists a correct node n_1, which never enters a higher view. Let v be the view n_1 is in. If any correct node ever enters a view higher than v, it sends a NewView message for that higher view to all nodes. n_1 will eventually receive this higher NewView message and enter a higher view, a contradiction. Thus, no node ever enters a view higher than v. Before entering view v, n_1 has sent $\langle$NewView, $v\rangle$ to all nodes. All correct nodes will eventually receive this $\langle$NewView, $v\rangle$ message, enter view v, and send $\langle$Status, $v, -\rangle$ messages. Eventually, correct nodes will receive $n-f$ $\langle$Status, $v, -\rangle$ messages and start their proposal timers. If n_1 receives $n-f$ $\langle$ViewChange, $v\rangle$ messages, it will enter view $v+1$, a contradiction. Thus n_1 never receives $n-f$ $\langle$ViewChange, $v\rangle$ messages. Then, there must be at least one correct node that never sends $\langle$ViewChange, $v\rangle$ and instead echoes $\langle$Propose, $v, -\rangle$ to all nodes. Eventually, all correct nodes will receive $\langle$Propose, $v, -\rangle$ message and send $\langle$Vote, $v, -\rangle$ messages to all nodes. Eventually n_1 will receive $n-f$ $\langle$Vote, $v, -\rangle$ messages and terminate, a contradiction. ◀

▶ **Theorem 12.** *All correct nodes eventually terminate.*

Proof. Suppose for the sake of contradiction that some correct node never terminates. Observe that if one correct node terminates, it sends a Commit message and makes all correct nodes eventually terminate. Thus, no correct node ever terminates. By lemma 11, every correct node keeps entering higher views.

Eventually, there will be a first time after $GST + 2d\Delta$ that some correct node enters a view v such that (i) the set F of crashed nodes no longer grows in views $\geq v$, (ii) $L_v \notin F$, and (iii) L_v is in the largest connected component $G' = (V - F, \diamond E)$. Let C denote this largest connected component. We next prove no node in C will ever send $\langle$ViewChange, $v\rangle$.

Let p be the first node in C that enters view v, and let p enter view v at time $t > GST+2d\Delta$. Observe that no node in C will send $\langle$ViewChange, $v\rangle$ before time $t + 3d'\Delta$ (proposal timer duration is $3d'\Delta$). Nodes in F crashed before entering view v and cannot send $\langle$ViewChange, $v\rangle$. Due to the condition in theorem 10, $n - |C \cup F| < n - f$. Thus, there will not be $n-f$ $\langle$ViewChange, $v\rangle$ messages before $t + 3d'\Delta$.

p sends $\langle$NewView, $v\rangle$ at time $t - 2d\Delta > GST$. All nodes in C receive $\langle$NewView, $v\rangle$ by $t - 2d\Delta + d'\Delta$, enter view v by $t + d'\Delta$, and stay in view v at least until $t + 3d'\Delta$.

When a node $q \in C$ receives $n-f$ $\langle$Status, $v, -\rangle$ messages at time $t' > t$, q echoes these $n-f$ messages and starts its proposal timer. All nodes in C enter view v by time $t + d'\Delta$ and are ready to echo these $\langle$Status, $v, -\rangle$ messages. (Recall that d' is the partially synchronous diameter of the graph.) L_v, which is in C, receives these $n-f$ $\langle$Status, $v, -\rangle$ messages by time $\max(t + 2d'\Delta, t' + d'\Delta) < t' + 2d'\Delta$. L_v sends a $\langle$Propose, $v, -\rangle$ message by time $t' + 2d'\Delta$ and it reaches q by time $t' + 3d'\Delta$, which is before q's proposal timer expires. Thus, q does not send $\langle$ViewChange, $v\rangle$. This establishes that no node in C will ever send $\langle$ViewChange, $v\rangle$. Again, nodes in F never send $\langle$ViewChange, $v\rangle$. Since $n - |C \cup F| < n - f$, there will never be $n-f$ $\langle$ViewChange, $v\rangle$ messages. Thus, no correct node ever enters a view higher than v. This contradicts lemma 11. ◀

5 BFT Consensus in Granular Partial Synchrony

▶ **Theorem 13.** *Under granular partial synchrony, BFT consensus with $n \geq 2f + 1$ on a graph G is solvable if and only if, for any set F of at most f faulty nodes, $\forall A \subseteq V - F$ with $|A| \geq n - 2f$, $\exists B \subseteq V - F$ with $|B| \geq f + 1$ such that $A \rightarrow B$.*

In words, the condition is that any honest set A of size at least $n - 2f$ has a potentially larger honest set B of size at least $f + 1$, such that for any node $b \in B$ there exits $a \in A$ and a synchronous path from a to b. Intuitively, if a message arrives at all of A, then it will also arrive at all of B after some delay.

Note that in BFT consensus, it never hurts the adversary to corrupt the maximum number of nodes allowed since Byzantine nodes can actively participate. This is why we can focus on the case of $|F| = f$ (as opposed to $|F| \leq f$).

Observe that the classic Byzantine fault tolerance bounds are special cases of our theorem. For example, when $n = 2f + 1$ and all links are synchronous, any $n - 2f = 1$ correct node has synchronous paths to all $n - f = f + 1$ correct nodes, so consensus is solvable. At the other extreme, $n = 3f + 1$ is the smallest value of n for which the condition in theorem 13 trivially holds even when all edges are partially synchronous (see necessity proof). And again, we will focus on the more interesting region of $2f + 1 < n \leq 3f$.

5.1 Necessary

The proof is again very similar to DLS [16]. The essence of the condition (and the proof) is to prevent a "split-brain" attack in which two groups of $n - 2f$ correct nodes cannot communicate in time and separately make progress with f Byzantine nodes.

Proof of Theorem 13 necessity part. For $n \geq 3f + 1$, the theorem is vacuous because the condition trivially holds: any set of $n - 2f \geq f + 1$ correct nodes have synchronous paths to at least $f + 1$ correct nodes (i.e., themselves).

For $n \leq 3f$, we prove by contradiction. Suppose there is an algorithm that solves consensus on a graph G that does not satisfy the condition in the theorem. Then, there exists a set F of f nodes such that, if nodes in F are faulty, a set A of $n - 2f$ correct nodes collectively have synchronous paths to at most f correct nodes. Let B be the set of these f nodes excluding A. Let C be the remaining nodes, i.e., $C = [n] \setminus \{F \cup A \cup B\}$. Note that $\{A, B, F, C\}$ is a four-way disjoint partition of the n nodes. Also note that $n - 2f = |A| \leq |A \cup B| \leq f$, $|F| = f$, and $|C| = n - |F \cup A \cup B| \leq f$.

Next, we consider three executions. In execution 1, all nodes have input v_1, and nodes in C are Byzantine. Since $|C| \leq f$, $A \cup B$ eventually decide v_1 in time t_1 due to validity. In execution 2, all nodes have input v_2, and nodes in $A \cup B$ are Byzantine. Since $|A \cup B| \leq f$, C eventually decide v_2 in time t_2 due to validity.

In execution 3, nodes in $A \cup B$ have input v_1, nodes in C have input v_2, nodes in F are Byzantine, and $GST > \max(t_1, t_2)$. F will behave towards $A \cup B$ like in execution 1 and towards C like in execution 2. Because there is no synchronous link between $A \cup B$ and C, $A \cup B$ cannot distinguish execution 3 from execution 1 and C cannot distinguish execution 3 from execution 2. Thus, $A \cup B$ decides v_1 and C decides v_2, violating agreement. ◀

5.2 Protocol

Next, we give a new BFT consensus protocol assuming the condition in theorem 13 holds. The protocol we present here achieves external validity [12]. In appendix C, we show how to extend it to achieve the strong unanimity validity in definition 4. This establishes the sufficiency of the condition.

Like in the CFT case, we will start from a standard leader-based partially synchronous BFT protocol and then take advantage of our graph condition to upgrade a quorum of $n-2f$ correct nodes to $f+1$ correct nodes.

A *lock* is a set L of $n-f$ signed matching $\langle$VOTE-1, $view, val\rangle$ messages from distinct nodes. Locks are ranked by their view numbers. We describe the protocol next.

Status step. Each view begins with every node sending a STATUS message to the leader of the current view. A node also starts a timer for the view.

Leader proposal step. When the leader of view v, L_v, receives a set S of $n-f$ $\langle$STATUS, $v, -\rangle$ messages from distinct nodes, it picks the highest-ranked lock among those. If no lock is reported, then the leader can safely propose its own input value, val_i. Otherwise, the leader must propose the value in the highest-ranked lock. The leader sends $\langle$PROPOSE, $v, val, S\rangle$ to all nodes. Note that a correct leader only sends one PROPOSE message in a view.

Equivocation check step. When a node receives $\langle$PROPOSE, $v, val, S\rangle$, it checks whether val is the highest-ranked locked value from the set S. If so, it forwards the PROPOSE message to all nodes and starts a timer for $d\Delta$ to listen for conflicting PROPOSE messages in the same view. If it receives a conflicting PROPOSE message, it detects the leader is faulty, forwards the equivocation to all nodes, and sends a VIEWCHANGE message for the current view. If the timer expires and no conflicting PROPOSE message is received, the node will send a $\langle$VOTE-1, $v, val\rangle$ message to all nodes indicating its support for the leader's proposal.

Locking step. When a node receives $n-f$ $\langle$VOTE-1, $v, val\rangle$ messages, it forms a lock certificate L for val in view v. The node updates its *lock* $:= L$ and sends a $\langle$VOTE-2, $v, val\rangle$ message to all nodes. The equivocation check guarantees the uniqueness of the locked value in each view.

Commit step. Upon receiving $C \leftarrow n-f$ $\langle$VOTE-2, $v, val\rangle$ messages, a node sends a $\langle$COMMIT, $C\rangle$ message. Upon receiving a $\langle$COMMIT, $C\rangle$ message, it commits and terminates.

View Change. A node sends $\langle$VIEWCHANGE, $v\rangle$ if it detects equivocation or times out in view v. Upon receiving $f+1$ VIEWCHANGE messages, a node stops sending VOTE-1/VOTE-2 messages in view v and sends its lock to all nodes. A node cannot immediately enter the next view but instead must wait $2d\Delta$ time before doing so. This is to give enough time for locks to propagate in the network.

5.3 Analysis

External validity is easily ensured if all correct nodes validate the proposed value before voting for it. In appendix C, we show how to achieve the strong unanimity validity in definition 4. We now focus on agreement and termination.

▶ **Lemma 14.** *If there exist $n-f$ $\langle$VOTE-1, $v, val\rangle$ messages and $n-f$ $\langle$VOTE-1, $v, val'\rangle$ messages in the same view v, then $val = val'$.*

Proof. Suppose for the sake of contradiction there exist a set S of $n-f$ $\langle$VOTE-1, $v, val\rangle$ messages and a set S' of $n-f$ $\langle$VOTE-1, $v, val'\rangle$ messages where $val \neq val'$.

Algorithm 3 BFT consensus protocol in granular partial synchrony for node i.

```
$v_i \leftarrow 0,\ lock \leftarrow \bot$                          ▷ Initialize local view number and lock
enter view 1

upon entering view $v$ do
    $v_i \leftarrow v$
    start $view\_timer \leftarrow timer((5+d)\Delta)$                ▷ Timer for changing view
    send $\langle$STATUS, $v, lock\rangle$ to $L_v$

upon receiving $S \leftarrow n-f$ $\langle$STATUS, $v_i, -\rangle$ do
    $val \leftarrow$ value in the highest lock in $S$, or $input_i$ if all locks in $S$ are $\bot$
    send $\langle$PROPOSE, $v_i, val, S\rangle$ to all

upon receiving $\langle$PROPOSE, $v_i, val, S\rangle$ from $L_{v_i}$ do
    if $val$ matches the highest locked value in $S$ or all locks in $S$ are $\bot$ then
        echo $\langle$PROPOSE, $v_i, val, S\rangle$ to all
        start $vote\_timer \leftarrow timer(d\Delta)$                ▷ To detect equivocation

upon $vote\_timer$ expiring and no equivocation detected do
    send $\langle$VOTE-1, $v_i, val\rangle$ to all

upon receiving $L \leftarrow n-f$ $\langle$VOTE-1, $v_i, val\rangle$ do
    $lock \leftarrow L$
    send $\langle$VOTE-2, $v_i, val\rangle$ to all

upon receiving $C \leftarrow n-f$ $\langle$VOTE-2, $v_i, val\rangle$ or one $\langle$COMMIT, $C\rangle$ do
    send $\langle$COMMIT, $C\rangle$ to all
    commit $val$ and terminate

upon receiving $\langle$PROPOSE, $v_i, val, -\rangle$ and $\langle$PROPOSE, $v_i, val', -\rangle$ where $val' \neq val$ do
    echo $\langle$PROPOSE, $v_i, val, -\rangle$ and $\langle$PROPOSE, $v_i, val', -\rangle$ to all
    send $\langle$VIEWCHANGE, $v_i\rangle$ to all

upon $view\_timer$ expiring do
    send $\langle$VIEWCHANGE, $v_i\rangle$ to all

upon receiving $VC \leftarrow f+1$ $\langle$VIEWCHANGE, $v\rangle$ where $v > v_i$ do
    stop sending VOTE-1/VOTE-2 messages for views up to $v$
    echo $VC$ to all
    echo $\langle$LOCKED, $lock\rangle$ to all
    wait $2d\Delta$
    enter view $v+1$

upon receiving $\langle$LOCKED, $lock'\rangle$ do
    $lock \leftarrow$ higher lock between $lock$ and $lock'$
    echo $\langle$LOCKED, $lock'\rangle$ to all
```

Of the $n-f$ nodes whose VOTE-1 messages are in S, at least a set P of $n-2f$ must be correct. By the condition in theorem 13, $P \rightarrow H$ where H is a set of $f+1$ correct nodes. Due to quorum intersection, $S' \cap H$ must contain at least one node, which is correct. Let c' be this node. Since the graph is undirected, there exists $c \in S$ such that $c' \rightarrow c$.

Let t be the time c' starts its vote timer. At time t, c' also forwards the $\langle$PROPOSE, $v, val', -\rangle$ message to all nodes. By time $t + d\Delta$, c receives this message. Thus, c must have sent $\langle$VOTE-1, $v, val\rangle$ before time $t + d\Delta$. Otherwise, c would have detected leader equivocation and would not have voted. Then, c must have forwarded $\langle$PROPOSE, $v, val, -\rangle$ to all nodes before time t. c' receives this $\langle$PROPOSE, $v, val, -\rangle$ message before time $t + d\Delta$, which is before its vote timer expires. Thus, c' detects leader equivocation and would not have voted. This contradicts $c' \in S'$. ◀

▶ **Lemma 15.** *If some node commits val in view v, then any set of $n-f$ $\langle$VOTE-1, $v', val'\rangle$ messages (lock certificate) in view $v' \geq v$ must have $val' = val$.*

Proof. We prove this lemma by induction on view v'. The base case of $v' = v$ is straightforward by lemma 14. For the inductive step, suppose the lemma holds up to view $v'-1$, and now we consider view v'. Suppose for the sake of contradiction that some node commits *val* in view v, and there exist $n-f > f$ nodes that send $\langle$VOTE-1, $v', val'\rangle$ messages for $val' \neq val$. A correct node will only send $\langle$VOTE-1, $v', val'\rangle$ if a proposal carries in view v' a set S of $\langle$STATUS, $v', -\rangle$ messages. Thus, there exists a subset $H \subseteq S$ of $n-2f$ correct nodes which sent $\langle$STATUS, $v', -\rangle$. By the condition in theorem 13, $H \rightarrow Q$, where Q is a set of $f+1$ correct nodes.

Since a node committed *val* in view v, there must exist some set $n-f$ nodes that sent $\langle$VOTE-2, $v, val\rangle$, of which a set R of at least $n-2f$ are correct. Before sending $\langle$VOTE-2, $v, val\rangle$ messages, these correct nodes updated $lock := (v, val)$ in view v. Sets Q and R intersect in at least one correct node. Let this node be q. Since the graph is undirected and $H \rightarrow Q$, there must exist a node $h \in H$ such that $q \rightarrow h$. By the induction hypothesis, any lock certificate from view v to $v'-1$ must be for *val*. Since a node only updates its lock monotonically based on view numbers, node q must have a lock with view $\geq v$ for *val*. Let t_h be the time node h echoed $f+1$ $\langle$VIEWCHANGE, $v'\rangle$ messages. By time $t_h + d\Delta$, node q must have received $f+1$ $\langle$VIEWCHANGE, $v'\rangle$ messages. Node q will then echo a $\langle$LOCKED, *lock*$\rangle$ message to all nodes. This will be received by node h by time $t_h + 2d\Delta$. Node h will update its lock to be at least view v for *val*. Thus, from nodes in H, $L_{v'}$ receives at least one STATUS message for *val* with view $\geq v$. By the induction assumption, any lock certificate not for *val* must have view $< v$. Thus, no correct node sends $\langle$VOTE-1, $v', val'\rangle$, a contradiction. ◀

▶ **Theorem 16** (Agreement). *No two correct nodes commit different values.*

Proof. Let v be the smallest view in which a correct node commits some value, say *val*. By lemma 15, only *val* can receive $n-f$ $\langle$VOTE-1, $v\rangle$ messages in any view $v' \geq v$, so no other value can be committed by a correct node. ◀

▶ **Theorem 17** (Termination). *All correct nodes eventually decide.*

Proof. With round-robin leader election, correct nodes are elected leaders infinitely often. Thus, there must be a view v, after $GST + 2d\Delta$, whose leader is correct. We next prove that all nodes will decide and terminate in view v (if they don't decide earlier).

Let t ($t \geq GST + 2d\Delta$) be the first time some correct node enters view v. This correct node echoes $f+1$ $\langle$VIEWCHANGE, $v-1\rangle$ messages to all nodes at $t - 2d\Delta \geq GST$. All correct nodes will receive the new view certificate by time $t - 2d\Delta + \Delta$, wait $2d\Delta$ themselves, and enter view v by time $t + \Delta$. Upon entering view v, they send $\langle \textsc{Status}, v, - \rangle$ messages to L_v. L_v receives $n - f$ $\langle \textsc{Status}, v, - \rangle$ messages by time $t + 2\Delta$, and send a $\langle \textsc{Propose}, v, - \rangle$ message to all nodes. All correct nodes will receive the $\langle \textsc{Propose}, v, - \rangle$ message by time $t + 3\Delta$ and start their vote timers. Since L_v is correct and does not equivocate, all correct nodes will send a $\langle \textsc{Vote-1}, v, - \rangle$ message by time $t + (3 + d)\Delta$. All correct nodes will receive $n - f$ $\langle \textsc{Vote-1}, v, - \rangle$ messages by time $t + (4 + d)\Delta$, and send a $\langle \textsc{Vote-2}, v, - \rangle$ message. All correct nodes will receive $n - f$ $\langle \textsc{Vote-2}, v, - \rangle$ messages and commit by time $t + (5 + d)\Delta$). Since a node's view timer is $(5 + d)\Delta$, and changing views requires $f + 1$ $\langle \textsc{ViewChange}, v \rangle$ messages, all correct nodes will remain in view v, commit and terminate in view v. ◀

6 Related Work

Necessary and sufficient conditions to solve consensus in all three classic timing models have been long established [27, 17, 15, 18, 9, 16]. There is also a large body of work on CFT and BFT consensus protocols in all three timing models. Our protocols adopt standard techniques from previous protocols such as quorum intersection [26, 29, 13], synchronous equivocation detection [23, 1, 2], and view synchronizers [11, 10].

Weaker models than synchrony have been suggested in the literature. Some of these are orthogonal to the timing model. A line of work studies consensus on *incomplete* communication graphs [34, 24, 25]. The mobile link failure model [33] allows a bounded number of lossy links. These models are orthogonal because they still need to adopt one of the classic timing models for the links that exist in the graph and are not lossy. The mobile sluggish model [19] allows temporary unbounded message delays for a set of honest nodes (the set can change over time). The sleepy model [30] allows a large fraction of nodes to be inactive. Both are models of node failures. Correct nodes that are not sluggish/sleepy are still assumed to have pair-wise synchronous links with each other.

The Visigoth fault tolerance (VFT) paper [31] proposes a timing model that consists of synchronous and asynchronous links. Their model assumes every node has asynchronous links to at most s correct nodes and synchronous links to the remaining nodes. For CFT, VFT requires $n - s \geq f + 1$, so every node must have at least $f + 1$ synchronous links. For BFT, VFT requires every node to have $n - s \geq 2f + 1$ synchronous links. In contrast, our graph conditions are weaker (less restrictive) in that they only require a set of $n - f$ nodes for CFT ($n - 2f$ correct nodes for BFT) to have synchronous paths to at least $f + 1$ nodes ($f + 1$ correct nodes for BFT). We additionally consider partially synchronous edges.

Another line of work that considers a mixture of links studies the minimal condition to circumvent the FLP [18] impossibility and solve consensus deterministically [20, 5, 8, 7, 4]. Many of these works [20, 5, 8] consider the harder setting of directed graphs, while we only consider undirected graphs. Since they focus on circumventing FLP, they only consider a mixture of asynchronous and partially synchronous links, but no synchronous links. Our main focus is to use synchronous links to achieve better fault tolerance than those under partial synchrony. But as mentioned, when $n > 2f$ for crash and $n > 3f$ for Byzantine, our "safety-critical" condition becomes vacuous, and our model degenerates to a mixture of partially synchronous and asynchronous links. In this context, our work establishes the minimum condition for circumventing FLP for CFT consensus in undirected graphs.

7 Conclusion

This paper introduces the granular synchrony model that considers a mixture of synchronous, partially synchronous, and asynchronous links to better capture the heterogeneity of modern networks. We present necessary and sufficient conditions for solving crash and Byzantine consensus in granular synchrony. Our results show that consensus is solvable in the presence of $f \geq n/2$ crash faults and $f \geq n/3$ Byzantine faults in granular synchrony, even though not all links are synchronous.

References

1 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118, 2020. doi:10.1109/SP40000.2020.00044.

2 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 331–341, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467899.

3 Saksham Agarwal, Qizhe Cai, Rachit Agarwal, David Shmoys, and Amin Vahdat. Harmony: A congestion-free datacenter architecture. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 329–343, Santa Clara, CA, April 2024. USENIX Association. URL: https://www.usenix.org/conference/nsdi24/presentation/agarwal-saksham.

4 Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 328–337, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1011767.1011816.

5 M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Consensus with byzantine failures and little system synchrony. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 147–155, 2006. doi:10.1109/DSN.2006.22.

6 Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Wpaxos: Wide area network flexible consensus. *IEEE Trans. Parallel Distrib. Syst.*, 31(1):211–223, January 2020. doi:10.1109/TPDS.2019.2929793.

7 Olivier Baldellon, Achour Mostéfaoui, and Michel Raynal. A necessary and sufficient synchrony condition for solving byzantine consensus in symmetric networks. In *International Conference on Distributed Computing and Networking*, pages 215–226. Springer, 2011. doi:10.1007/978-3-642-17679-1_19.

8 Zohir Bouzid, Achour Mostfaoui, and Michel Raynal. Minimal synchrony for byzantine consensus. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 461–470, 2015. doi:10.1145/2767386.2767418.

9 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

10 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Liveness and Latency of Byzantine State-Machine Replication. In Christian Scheideler, editor, *36th International Symposium on Distributed Computing (DISC 2022)*, volume 246 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2022.12.

11 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. *Distributed Computing*, 35(6):503–532, 2022. doi:10.1007/S00446-022-00432-Y.

12 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography (extended abstract). In

Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '00, pages 123–132, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/343477.343531.

13 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, USA, 1999. USENIX Association. URL: https://dl.acm.org/citation.cfm?id=296824.

14 Shinyoung Cho, Romain Fontugne, Kenjiro Cho, Alberto Dainotti, and Phillipa Gill. Bgp hijacking classification. In *2019 Network Traffic Measurement and Analysis Conference (TMA)*, pages 25–32, 2019. doi:10.23919/TMA.2019.8784511.

15 D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, November 1983. doi:10.1137/0212045.

16 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988. doi:10.1145/42282.42283.

17 Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '85, pages 59–70, New York, NY, USA, 1985. Association for Computing Machinery. doi:10.1145/323596.323602.

18 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.

19 Yue Guo, Rafael Pass, and Elaine Shi. Synchronous, with a chance of partition tolerance. In *Advances in Cryptology – CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I*, pages 499–529, Berlin, Heidelberg, 2019. Springer-Verlag. doi:10.1007/978-3-030-26948-7_18.

20 Moumen Hamouma, Achour Mostéfaoui, and Gilles Trédan. Byzantine consensus with few synchronous links. In *Principles of Distributed Systems: 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings 11*, pages 76–89. Springer, 2007. doi:10.1007/978-3-540-77096-1_6.

21 Owen Hilyard, Bocheng Cui, Marielle Webster, Abishek Bangalore Muralikrishna, and Aleksey Charapko. Cloudy forecast: How predictable is communication latency in the cloud?, 2023. arXiv:2309.13169, doi:10.48550/arXiv.2309.13169.

22 Toke Høiland-Jørgensen, Bengt Ahlgren, Per Hurtig, and Anna Brunstrom. Measuring latency variation in the internet. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 473–480, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2999572.2999603.

23 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006*, pages 445–462, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. doi:10.1007/11818175_27.

24 Muhammad Samir Khan, Syed Shalan Naqvi, and Nitin H. Vaidya. Exact byzantine consensus on undirected graphs under local broadcast model, 2019. arXiv:1903.11677.

25 Muhammad Samir Khan and Nitin Vaidya. Asynchronous byzantine consensus on undirected graphs under local broadcast model, 2019. arXiv:1909.02865.

26 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. doi:10.1145/279227.279229.

27 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982. doi:10.1145/357172.357176.

28 Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2517349.2517350.

29 Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM*

Symposium on Principles of Distributed Computing, PODC '88, pages 8–17, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/62546.62549.

30 Rafael Pass and Elaine Shi. The sleepy model of consensus. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 380–409, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-70697-9_14.

31 Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2741948.2741979.

32 Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. SwiftPaxos: Fast Geo-Replicated state machines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 345–369, Santa Clara, CA, April 2024. USENIX Association. URL: https://www.usenix.org/conference/nsdi24/presentation/ryabinin.

33 U. Schmid, B. Weiss, and J. Rushby. Formally verified byzantine agreement in presence of link faults. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 608–616, 2002. doi:10.1109/ICDCS.2002.1022311.

34 Lewis Tseng and Nitin Vaidya. Exact byzantine consensus in directed graphs, 2014. arXiv:1208.5075.

35 Tian Yang, Robert Gifford, Andreas Haeberlen, and Linh Thi Xuan Phan. The synchronous data center. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 142–148, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3317550.3321442.

A BFT Consensus in Granular Asynchrony

We present a sufficient condition for solving BFT consensus in granular asynchrony.

▶ **Theorem 18.** *If (i) the condition in theorem 13 holds and (ii) for all F with $|F| = f$, there exists a node in graph $G' = (V - F, \diamond E)$, which has partially synchronous paths to f other nodes in G', then BFT consensus on graph $G = (V, E)$ can be solved deterministically under granular asynchrony.*

We have proved the necessity of condition (i) (for all algorithms) in Section 5.1. Condition (ii) was proven necessary in [7] for algorithms that work for the family of all graphs that satisfy the condition (i.e., graph-agnostic algorithms). If algorithms can be tailored to the graph, the tight condition for Byzantine consensus remains open.

A.1 Protocol

Next, we adapt our previous BFT consensus protocol in algorithm 3 from granular partial synchrony to granular asynchrony, assuming the condition in theorem 18 holds. This establishes the sufficiency of the condition.

As with our granular asynchrony CFT consensus protocol, we utilize condition (ii) in theorem 10 to guarantee that, when the correct node with partially synchronous paths to f other nodes in $G' = (V - F, \diamond E)$ is elected leader after GST, this leader will not be replaced and will make progress. To do so, we first require $n - f$ nodes to initiate a view change, of which at least $n - 2f$ must be correct. This way, because fewer than $n - 2f$ correct nodes are asynchronously connected to the leader, we just need to make sure that none of the f nodes the leader is connected to via partially synchronous paths initiates a view change.

We only describe the status and view change steps, since the rest of the protocol remains the same as algorithm 3.

Algorithm 4 BFT consensus protocol in granular asynchrony for node i.

```
$v_i \leftarrow 0$, $lock \leftarrow \bot$                         ▷ Initialize local view number and lock
enter view 1

upon entering view $v$ do
    $v_i \leftarrow v$
    send $\langle$STATUS, $v, lock\rangle$ to all

upon receiving $n - f$ $\langle$STATUS, $v_i, -\rangle$ messages where $i \neq L_v$ do
    echo these $n - f$ $\langle$STATUS, $v_i, -\rangle$ to all
    start $proposal\_timer \leftarrow timer(3d'\Delta)$                ▷ Timer before changing view

upon $proposal\_timer$ expiring and no leader proposal received do
    send $\langle$VIEWCHANGE, $v_i\rangle$ to all

PROPOSAL, VOTE-1, VOTE-2, COMMIT messages, $n-f$ VIEWCHANGE messages (instead
of $f + 1$), equivocation detection at all nodes, and STATUS messages at view leaders are
processed the same way as in Algorithm 3
```

Status step. Upon entering a new view v, a node sends a $\langle$STATUS, $v, lock\rangle$ message to **all** nodes. When a node receives at least $n - f$ $\langle$STATUS, $v, -\rangle$ messages, it forwards this set of STATUS messages to all nodes and starts a timer with $3d'\Delta$ duration. The same propose, vote, and commit steps from algorithm 3 follow.

View change. A node suspects the leader is faulty if it does not receive a $\langle$PROPOSE, $v, -, -\rangle$ message before its proposal timer (instead of view timer) expires. A view change certificate consists of $n - f$ $\langle$VIEWCHANGE, $v\rangle$ messages (instead of $f + 1$ in algorithm 3). Upon receiving $n - f$ $\langle$VIEWCHANGE, $v\rangle$, a node carries out the same waiting period step from algorithm 1.

A.2 Analysis

The agreement and validity proofs are identical to the granular partial synchrony BFT case. We focus on termination.

▶ **Lemma 19.** *If no correct node ever terminates, then every correct node keeps entering higher views.*

Proof. Suppose for the sake of contradiction, there exists a correct node n_1, which never enters a higher view. Let v be the view n_1 is in. If any correct node ever enters a view $v' > v$, it must have echoed $n - f$ $\langle$VIEWCHANGE, $v' - 1\rangle$ messages to all nodes. n_1 will eventually receive this set $n - f$ $\langle$VIEWCHANGE, $v' - 1\rangle$ messages and enter a higher view, a contradiction. Thus, no correct node ever enters a view higher than v. Before entering view v, n_1 must have sent $n - f$ $\langle$VIEWCHANGE, $v - 1\rangle$ to all nodes. All correct nodes will eventually receive this set of $\langle$VIEWCHANGE, $v - 1\rangle$ messages, enter view v, and send a $\langle$STATUS, $v, -\rangle$ message. Eventually, correct nodes will receive $n - f$ $\langle$STATUS, $v, -\rangle$ messages and start their proposal timers. If n_1 receives $n - f$ $\langle$VIEWCHANGE, $v\rangle$ messages, it will enter view $v + 1$, a contradiction. Thus n_1 never receives $n - f$ $\langle$VIEWCHANGE, $v\rangle$ messages. Then, there must be at least one correct node, n_2, which never sends $\langle$VIEWCHANGE, $v\rangle$, and instead echoes $\langle$PROPOSE, $v, -, -\rangle$ to all nodes. Eventually, all correct nodes will receive a ⟨Propose, $v, -, -$⟩ message and echo it. If a correct node detects leader equivocation, it will forward it to all correct nodes. n_2 will eventually receive the conflicting Propose messages and send a ⟨ViewChange, v⟩ message, a contradiction. Thus, no correct node will detect leader equivocation. Then, all correct nodes will send ⟨Vote-1, $v, -$⟩ messages to all nodes. Eventually all correct nodes will receive $n - f$ ⟨Vote-1, $v, -$⟩ messages, and send a ⟨Vote-2, $v, -$⟩ message. Eventually, n_1 will receive $n - f$ ⟨Vote-2, $v, -$⟩ messages, commit and terminate, a contradiction. ◀

▶ **Theorem 20.** *All correct nodes eventually terminate.*

Proof. Suppose for the sake of contradiction that some correct node never terminates. Observe that if one correct node terminates, it sends a Commit message and makes all correct nodes eventually terminate. Thus, no correct node ever terminates. By lemma 19, every correct node keeps entering higher views.

Eventually, there will be a first time after $GST + 2d\Delta$ that some correct node enters a view v such that (i) $L_v \notin F$, and (ii) L_v has paths to at least f other nodes in graph $G' = (V - F, \diamond E)$. Let C denote this set of nodes including L_v. We next prove no node in C will ever send ⟨ViewChange, v⟩.

Let p be the first node in C that enters view v, and let p enter view v at time $t > GST{+}2d\Delta$. Observe that no node in C will send ⟨ViewChange, v⟩ before time $t + 3d'\Delta$ (proposal timer duration is $3d'\Delta$). Due to the condition in theorem 18, $n - |C| < n - f$. Thus, there will not be $n - f$ ⟨ViewChange, v⟩ messages before $t + 3d'\Delta$.

p sends $n - f$ ⟨ViewChange, $v - 1$⟩ messages at time $t - 2d\Delta > GST$. All nodes in C receive $n - f$ ⟨ViewChange, $v - 1$⟩ messages by time $t - 2d\Delta + d'\Delta$, enter view v by time $t + d'\Delta$, and stay in view v at least until time $t + 3d'\Delta$.

When a node $q \in C$ receives $n - f$ ⟨Status, $v, -$⟩ messages at time $t' > t$, q echoes these $n - f$ messages and starts its proposal timer. All nodes in C enter view v by time $t + d'\Delta$ and are ready to echo these ⟨Status, $v, -$⟩ messages by $t + d'\Delta$. L_v, which is in C, receives these $n - f$ ⟨Status, $v, -$⟩ messages by time $\max(t + 2d'\Delta, t' + d'\Delta) < t' + 2d'\Delta$. L_v sends a ⟨Propose, $v, -$⟩ message by time $t' + 2d'\Delta$ and it reaches q by time $t' + 3d'\Delta$, which is before q's proposal timer expires. Thus, q does not send ⟨ViewChange, v⟩. This establishes that no node in C will ever send ⟨ViewChange, v⟩.

Since $n - |C| < n - f$, there will never be $n - f$ ⟨ViewChange, v⟩ messages. Thus, no correct node ever enters a view higher than v. This contradicts lemma 19. ◀

B Comparison with [4]

[4] showed that a correct $\diamond f$-source is a sufficient condition for solving CFT consensus in a directed graph. A correct $\diamond f$-source is a correct node that has f outgoing fault-free paths that are eventually synchronous. [4] argued the potential optimality of their result by showing that every node being a $\diamond(f-1)$-source is not sufficient for solving CFT consensus. Our results show that, at least in the case of undirected graphs, a correct $\diamond f$-source is not necessary. Our condition (ii) in theorem 10 is weaker and is sufficient.

To show our condition is weaker, we first prove that a correct $\diamond f$-source implies the condition (ii) in theorem 10. Let C be the connected component in $G' = (V, \diamond E)$ that the correct $\diamond f$-source belongs to. We have $|C| \geq f + 1$. Removing $F \cup C$ must leave at most $n - f - 1$ nodes in the remaining graph.

Next, Figure 2 shows an example of a graph that satisfies our condition but does not have a correct $\diamond f$-source. For this graph, if the adversary corrupts B and C, then there is no correct $\diamond f$-source since A only has a link to B and D only has a link to C. This graph,

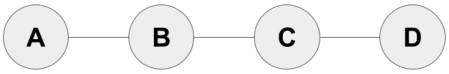


Figure 2 In this graph $n = 4$ and $f = 2$. Each edge represents a synchronous link and a missing edge represents an asynchronous link.

Algorithm 5 BFT Unanimity Validity.

```
$v_i \leftarrow 0$, $lock \leftarrow \bot$          ▷ Initialize local view number and lock
$inputs \leftarrow \{\}$

echo ⟨INPUT, $input_i$⟩ to all
start $input\_timer \leftarrow timer(2d\Delta)$

upon receiving $m \leftarrow$ ⟨INPUT, $input_j$⟩ do
    echo $m$
    $inputs \leftarrow inputs \cup \{m\}$

upon $input\_timer$ expiring do
    send ⟨FORWARD-INPUTS, $inputs$⟩ to all

upon receiving $FI \leftarrow n - f$ ⟨FORWARD-INPUTS, $inputs$⟩ do
    if having received $I \leftarrow f + 1$ ⟨INPUT, $val$⟩ messages in $FI$ then
        $lock \leftarrow I$

enter view 1
```

however, satisfies the condition (ii) in theorem 10. If $|F| = 0$, removing the largest connected component (the entire graph) leaves 0 nodes, satisfying the condition. For any choice of F with $|F| = 1$, the largest connected component after removing F must be of size at least 2. Thus, there will be at most 1 remaining node, satisfying the condition. For any choice of F such that $|F| = 2$, the largest remaining connected component must be of size at least 1. Thus, there will be at most 1 remaining node, satisfying the condition.

C BFT Unanimity Validity

In this section, we give a way to convert our BFT algorithms from external validity to strong unanimity validity. The idea is to try to have nodes lock before starting the first view, and if all correct nods have the same input, then that input is the only lock.

▶ **Lemma 21.** *If all correct nodes have the same input, then all correct nodes will lock on this value before entering view* 1*, and any lock in view* 0 *must be for val.*

Proof. In view 0, all correct nodes send their inputs and echo other nodes' inputs they receive (using INPUT and FORWARD-INPUTS messages) before their input timer expires in $2d\Delta$ time. For any two correct nodes p and q such that $p \rightarrow q$, p will receive q's input before p's input timer expires. Similarly, q will receive p's input before q's input timer expires. Consider any correct node c. Node c will eventually receive a set A of $n - f$ ⟨FORWARD-INPUTS, $inputs$⟩ messages. Among them, a subset B of $n - 2f$ are from correct nodes. By the condition in theorem 13, $B \rightarrow C$ where C is a set of $f + 1$ correct nodes. Since every node in B waits $2d\Delta$ before sending a FORWARD-INPUTS message, this is sufficient time for each node in C

to receive an input from some node in B and also sends its input to that node in B. Thus, B will contain the input values from C, a set of $f+1$ correct nodes. If all correct nodes have the input *val*, node c must receive at least $f+1$ $\langle$INPUT, *val*$\rangle$ messages, and there are at most f INPUT messages for a different value (from f Byzantine nodes). Therefore, every correct node will set its lock to $I \leftarrow f+1$ $\langle$INPUT, *val*$\rangle$ in view 0, and any lock in view 0 must be for *val*. ◀

▶ **Lemma 22.** *If all correct nodes have the same input, then any lock in view $v \geq 0$ must be for val.*

Proof. The base case is established by lemma 21. Now assume the lemma holds for all $v-1$, and consider view v. Suppose for the sake of contradiction a lock forms for $val' \neq val$. L_v must have proposed $val' \neq val$. By the induction assumption, any lock must be for *val*. Thus, L_v must have received $S \leftarrow n-f$ STATUS messages where all locks are $\perp$. By lemma 21, all correct nodes will lock on *val* before entering view 1. The set S must contain a STATUS message from at least one correct node. This correct node will at least have a lock in view 0 or higher, and thus its STATUS message will not have $lock = \perp$, a contradiction. ◀

▶ **Theorem 23.** *If all correct nodes have the same input, then only that value can be decided.*

Proof. By lemma 22, any lock must be for *val*, the input of the correct nodes. Only locked values can be decided. Validity then follows from termination. ◀

Distributed Delta-Coloring Under Bandwidth Limitations

Magnús M. Halldórsson
Reykjavik University, Iceland

Yannic Maus
TU Graz, Austria

Abstract

We consider the problem of coloring graphs of maximum degree Δ with Δ colors in the distributed setting with limited bandwidth. Specifically, we give a $\text{poly}\log\log n$-round randomized algorithm in the CONGEST model. This is close to the lower bound of $\Omega(\log\log n)$ rounds from [Brandt et al., STOC '16], which holds also in the more powerful LOCAL model. The core of our algorithm is a reduction to several special instances of the constructive Lovász local lemma (LLL) and the $deg + 1$-list coloring problem.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Graph problems, Graph coloring, Lovász local lemma, LOCAL model, CONGEST model, Distributed computing

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.31

Related Version *Full Version*: https://arxiv.org/abs/2405.09975

Funding *Magnús M. Halldórsson*: Supported in part by Icelandic Research Fund grant 217965.
Yannic Maus: Supported by the Austrian Science Fund (FWF), Grant P36280-N.

Acknowledgements We thank Saku Peltonen for valuable input and discussions.

1 Introduction

The objective in the c-coloring problem is to color the vertices of a graph with c such that any two adjacent vertices receive different colors. In the distributed setting, the $\Delta + 1$-coloring problem has long been the focus of interest as the natural *local* coloring problem: any partial solution can be extended to a valid full solution. It has fast $\text{poly}(\log\log n)$-round algorithms, both in LOCAL [12] and CONGEST [29], and so does the more general *deg+1*-list coloring problem (d1LC), which is what remains when a subset of the nodes has been $\Delta + 1$-colored [30, 34].

The Δ-coloring problem, on the other hand, is *non-local*: fixing the colors of just two nodes can make it impossible to form a proper Δ-coloring, see Figure 1 for an example. Due to its simplicity, it has become the prototypical problem for the frontier of the unknown [27, 2]. Even the existence of such colorings is non-trivial: a celebrated result by Brooks from the '40s shows that Δ-colorings exist for any connected graph that is neither an odd cycle nor a clique on $\Delta + 1$ nodes [10].

A $\text{poly}(\log\log n)$-round Δ-coloring algorithm was recently given in LOCAL [22], but no non-trivial algorithm is known in CONGEST. It is of natural interest to examine if the transition from local to non-local problems behaves differently in LOCAL and in CONGEST. Thus, we set out to answer the following question:

> Is there a sublogarithmic time distributed Δ-coloring algorithm using small messages?

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 31; pp. 31:1–31:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this work, we answer the question in the affirmative. We prove the following theorem.

▶ **Theorem 1.** *There is a randomized* $\operatorname{poly}\log\log n$*-round* CONGEST *algorithm to* Δ*-color any graph with maximum degree* $\Delta \geq 3$*. The algorithm works with high probability.*

Theorem 1 nearly matches the lower bound of $\Omega(\log\log n)$ that holds in LOCAL [9]. In [2], the authors claim that in order to make progress in our understanding of distributed complexity theory, we require a Δ-coloring algorithm that is genuinely different from the approaches in [41, 27]. This is due to the fact that the current state-of-the-art runtime for Δ-coloring lies exactly in the regime that is poorly understood. The approaches of [41, 27] are based on brute-forcing solutions on carefully chosen subgraphs of super-constant diameter. In contrast, our results are based on a bandwidth-efficient deterministic reduction to a constant number of "simple" Lovász Local Lemma (LLL) instances and $O(\log\Delta)$ instances of d1LC; the LLL is a general solution method applicable to a wide range of problems. It is known that LLL is complete for sublogarithmic computation on constant-degree graphs, but its role on general graphs is widely open [13]. Our algorithm adds to the small list of problems (see the related work section in [33]) that can be solved in sublogarithmic time with an LLL-type approach, even under the presence of bandwidth restrictions. Before continuing further, let us first detail the computational model.

In the CONGEST model, a communication network is abstracted as an n-node graph of maximum degree Δ, where nodes serve as computing entities and edges represent communication links. Initially, a node is unaware of the topology of the graph G, nodes can communicate with their neighbors in order to coordinate their actions. This communication happens in synchronous rounds where, in each round, a node can perform arbitrary local computations and send one message of $O(\log n)$ bits over each incident edge. At the end of the algorithm, each node outputs its own portion of the solution, e.g., its color in coloring problems. The LOCAL model is identical, except without restrictions on message size.

1.1 Technical Overview on Previous Approaches

Previous fast distributed Δ-coloring algorithms either use huge bandwidth [41, 27] or use limited bandwidth but only work in the extreme cases of either very high-degree [22] or super low-degree graphs [40]. Optimally, we would like to take any of these solutions and run them with minor modifications to obtain an algorithm that uses low bandwidth and works for all degrees. This approach is entirely infeasible for the highly specialized algorithms in [41, 27, 28]. These works crucially rely on learning the full topology of non-constant diameter subgraphs, which is impossible in CONGEST.

For graphs of super-low degree, i.e., at most $\operatorname{poly}\log\log n$, an efficient Δ-coloring algorithm with low bandwidth can be deduced from the results in [40]. In fact, the paper takes a complexity-theoretic approach and shows that any problem can be solved in sublogarithmic time with low bandwidth as long as 1) the problem is defined on low-degree graphs, 2) a given solution can be checked efficiently for correctness by a distributed algorithm, and 3) the problem admits a sublogarithmic time LOCAL model algorithm. As such, the results are not very constructive for any specific problem like the Δ-coloring problem. In fact, it is known that these generic techniques cannot be extended to problems defined on graphs with larger degrees [3], which is the main target of our work.

Our best hope is then the $\operatorname{poly}\log\log n$-round LOCAL model algorithm of [22]. We discuss it in detail throughout the next few pages as it motivates the design choices of our solution. Unfortunately, for maximum degrees that are at most poly-logarithmic, it relies on the prior $O(\log\Delta) + \operatorname{poly}\log\log n$-round LOCAL model algorithm from [27] in a black-box manner.

For large maximum degrees, however, when Δ is $\omega(\log^3 n)$, they provide a sophisticated constant-round randomized reduction to the *deg* + 1-list coloring problem (d1LC) that also works with low bandwidth. The central ingredient in this reduction is the notion of slack.

Slack. To reduce the Δ-coloring problem to d1LC, it suffices to obtain a unit amount of *slack* for each node. Namely, if two neighbors of a node are assigned the same color, there are then more colors available to the node than its number of uncolored neighbors. Slack can be easily generated w.h.p. (for most, but not all, kinds of nodes) with a simple single-round procedure termed SlackGeneration, as long as the graph has high degree. This observation has been used in countless papers on various coloring problems, e.g., [18, 35, 12, 29, 22]. For intermediate-degree graphs, this slack generation problem can be formulated as an instance of the constructive Lovász Local Lemma (LLL), but one that seems inherently non-implementable in CONGEST, as we explain later.

Recall that the LLL is a general solution method applicable to a wide range of problems. Defined over a set of independent random variables, it asks for an assignment of the variables that avoids a set of "bad" events. The original theorem [19] shows that such an assignment exists as long as the probability of the events to occur is sufficiently small in relation to the dependence degree of the events, i.e., the number of other events that share a variable. There is now a general LOCAL algorithm running in $O(\log n)$ rounds of LOCAL [39, 15], but superfast poly($\log\log n$) algorithms are only known for restricted cases [21, 26, 17]. Even less is known about solvability in CONGEST [31, 33].

In the presented slack generation LLL, there is a bad event for each node that holds if the respective node does not obtain slack. The mentioned SlackGeneration works as follows. Each node gets activated with a constant probability, picks a random candidate color that it keeps if no neighbor wants to get the same color and discards otherwise (see Algorithm 3 in Section 3 for details). Hence, there are random variables for each node depicting its activation status and candidate color choice. The main reason why this LLL cannot be directly implemented in CONGEST is that events involve values of variables at distance 2 in the communication graph. This makes it impossible for an event node to obtain full information on the status of all its variables, an ingredient that essentially is crucial in all known sublogarithmic-time LLL algorithms. The formal meaning of the word "essential" in that sentence is extremely technical and is captured by the notion of a *simulatable* LLL (see the full version of this paper). In essence, it says that the LLL is easy enough such that event nodes can learn enough information about their variables to execute some simple primitives such as evaluating their status (does the event hold or not), resampling their variables, and computing certain conditional probabilities for the event to hold under partial variable assignments. The latter condition is the most challenging one to ensure.

1.2 Our Technical Approach

What we have discussed so far is only half the truth. In fact, the slack generation process only works for *sparse* nodes, i.e., nodes with many non-edges in their neighborhood. If the graph is locally too dense, then slack cannot be obtained via this LLL. Thus, the algorithm of [22] carefully analyzes the topological structure of the hard instances for Δ-coloring, combining several different (deterministic and randomized) methods to create slack. Such a treatment seems to be inherent to the Δ-coloring problem as a very similar classification was independently and currently discovered in the streaming model [1]. Additionally, it has also been shown to be useful in different models of computation. In the aftermath of these works, it has been used to obtain efficient massively parallel algorithms for the problem [16].

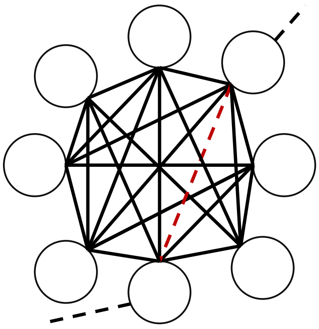


Figure 1 This is an example of an almost clique (AC). The depicted nice AC is a clique on $\Delta + 1$ nodes with a single missing (red) edge. It is essential that the two nodes incident to the missing edge receive the same color to solve the Δ-coloring problem. All non-nice ACs form proper cliques.

Our algorithm is based on a fine-grained version of this classification equipped with a sequence of various LLLs for eventual slack generation. Each LLL is easier to solve in the CONGEST model than the aforementioned slack generation LLL. In the following, we use the terminology of [22], and explain their algorithm and our solution in more detail.

Like in all recent randomized distributed graph coloring algorithms, they divide the graph into sparse and dense parts that are referred to as "almost-cliques" (ACs). Then, they partition the ACs further into different types – *ordinary, nice, difficult* – each of which admits a different coloring approach. See Figure 1 for an example of an AC. One challenge is that all these different types of tricky subgraphs may appear in the same graph and close to each other. For this overview it is best to imagine each AC as a proper clique on almost Δ nodes in which each node has a few external neighbors residing in other ACs and creating lots of dependencies between different ACs. Thus, their algorithm is fragile with regard to the order in which different types of ACs are colored. The starting point of our work is that the core step of their algorithm does not work in low-degree graphs. More detailed, the first step of their algorithm executes SlackGeneration (see Algorithm 3 in Section 3) on a carefully selected subset of nodes to achieve three objectives: **a)** giving slack to all sparse nodes, **b)** providing a slack-toehold[1] for a subclass of the difficult ACs that the authors term "runaway", and **c)** providing each ordinary clique with a node that has slack. Each of these probabilistic guarantees holds w.h.p. as long as $\Delta = \omega(\log^3 n)$. Their proof shows that, in essence, all three cases are LLLs but ones that are far from being simulatable. We discuss our solutions for a)–c), separately.

Solution for a). Providing slack to sparse graphs is the main application of the LLL algorithm in [33]. In essence, we adapt their techniques to provide slack to sparse nodes but provide additional guarantees that are needed for other parts of the graph.

Solution for b). For the difficult cliques we propose a solution that eliminates randomness and solely colors all the nodes via a sequence of d1LC instances. See Figure 2 for an illustration of our solution. First, we adjust the classification of difficult almost-cliques from [22]. All nodes in a given difficult clique have the same external degree. We associate with each such AC C a *special node* s_C on its outside that has many neighbors on the inside (namely, more than twice the external degree of C's nodes).

[1] A slack-toehold for an AC is an uncolored node that can be stalled to be colored later. All of its neighbors then lose one competitor for the remaining colors, providing them with temporal slack.

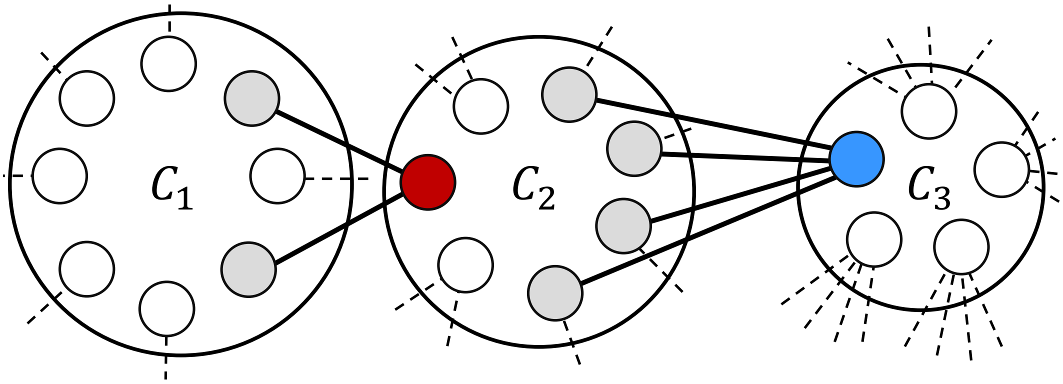


Figure 2 for part b): The illustration depicts three difficult cliques of different layers. The external degree of C_1 is 1, the external degree of C_2 is 2 and the external degree of C_3 is 4. C_1 has the lowest layer and its special node (the red node) is part of C_2. The blue special node of C_2 is part of C_3. So when we color C_1 the red node serves as an uncolored toehold providing slack to two gray nodes of C_1. Stalling the coloring of these gray nodes provides slack to the white nodes of C_1 so that they can be colored, followed by the gray ones. For illustration purposes, we chose Δ to be 9, but note that this would actually not classify C_3 as a difficult clique. A special node of C_3 would need $2e_{C_3} = 8$ neighbors in C_3, which is impossible due to C_3's size.

From here, we assign each difficult clique a *layer* that determines the step in which it gets colored. Those with a special node that is *not* contained in another difficult clique are treated separately and assigned to layer ∞, to be dealt with at the very end. The other difficult cliques are assigned to layers indexed by the base-2 logarithm of their external degree. The crucial property that follows is that the cliques in a given layer have their special node in a higher layer. This allows us to color the cliques layer by layer, starting with smaller layers. The special node s_C is stalled to be colored later, providing a toehold for C. This way, we color the cliques and special nodes in all layers besides ∞.

This leaves the problem of coloring ACs the ∞ layer and their still uncolored special nodes. In this exposition, we assume that special nodes are not shared by multiple difficult cliques. In that case, we pair the special node s_C up with some node $u_C \in C$ that is not adjacent to s_C with the objective to *same-color* the nodes: assigning both the same color. This is done via a virtual coloring problem capturing the dependencies between all selected pairs in the participating difficult cliques and the restrictions imposed by already colored vertices of the graph. We show that this virtual coloring instance is indeed a d1LC instance and can be solved efficiently in CONGEST despite being a problem on a virtual graph. As a result, the clique C obtains an uncolored node y_C that is adjacent to both s_C and u_C, has slack due to two same-colored neighbors, and can serve as a toehold for C.

Besides removing the need for randomization to solve the difficult cliques, our classification of difficult cliques also captures significantly more ACs than the definition of difficult cliques in [22]. The additional structure provided to the remaining ACs is exploited down the line in the most challenging part of the algorithm, dealing with the ordinary cliques in part c).

Solution for c). The most involved part by far is dealing with case c). We split the ordinary cliques into the small (of size less than $\Delta - \Delta/\operatorname{poly}\log\log(n)$) and large. The small ones can be handled just like the sparse nodes, as one can show that their induced neighborhoods are relatively sparse. The main effort then is to manually create slack for the large ordinary cliques. For this exposition, it is best to imagine an ordinary clique to be a clique on Δ nodes in which each node of the clique has exactly one external neighbor that is again a member of a large ordinary clique. See Figure 3 for an illustration.

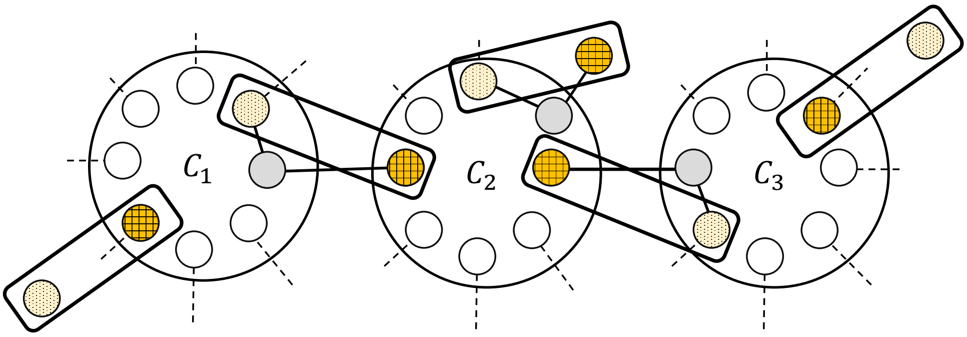


Figure 3 for Part c): For the large ordinary cliques, we find triples of nodes consisting of a yellow (striped), a light yellow (dotted), and a gray (solid) node. The two yellow nodes are non-adjacent while the gray node is adjacent to both of them. The goal is to same-color the pairs of yellow/light-yellow nodes, to which end we form a virtual coloring instance consisting of all pairs and their dependencies. After same-coloring the yellow nodes, the gray node provides a slack-toehold for the clique. An important aspect is that triples of different ordinary ACs are non-overlapping and no neighborhood of the graph contains too many nodes in such pairs, as otherwise we may run into unsolvable subinstances down the line. We find these triples by a sequence of "simple" LLLs.

In order to create slack-toehold in each large AC C, we compute a "vee-shaped" triple (x_C, y_C, z_C) of nodes, with $x_C, y_C \in C$ and $z_C \notin C$, but $z_C \in N(y_C)$ and z_C is also a non-neighbor of x_C. Then, we set up a virtual list coloring instance with a node for each such pair with the objective to same-color the pairs (x_C, z_C). As we ensure that y_C is uncolored, it serves as a slack-toehold for the AC. As many of the important ACs can be mutually adjacent, the main difficulty lies in finding non-overlapping triples for the ACs. We ensure this by first computing a suitable candidate set Z from which we then pick the third node z_C of the triple. Finding the set Z can be modeled as an "easy" LLL fitting the framework of [33]. Finding the node $z_C \in Z$ can also be modeled as a different type of "easy" LLL. In essence, the first LLL is easy (in CONGEST) as its bad events only consist of simple bounds on the number of neighbors in Z. Next, we elaborate on our LLL for finding $z_C \in Z$ with slightly more detail; due to further technicalities of the existing LLL algorithms from which we spare you in this technical overview, our actual solution differs slightly from the one presented here.

With a given set Z, we model the problem of selecting $z_C \in Z$ as an LLL as follows. Each AC C sends a proposal (to serve as its z_C node) to each outside neighbor inside Z with probability $\operatorname{poly}\log\log n/\Delta$. The proposal is *successful* if no other AC proposes to that node. We show that with a constant probability, no other AC proposes to the same node and that this is independent for different nodes in Z. Since we ensure C has many neighbors in Z, we obtain that the probability that none of C's proposals are successful is bounded above by $p = \exp(-\Omega(\operatorname{poly}\log n))$. The main benefit is that this LLL and also the LLL for finding the set Z are simple enough to be simulatable (in contrast to LLLs based on randomized slack generation for those ACs that can be derived from the proofs in [33]).

Once we have found z_C, the structure of large ordinary ACs implies that we can deterministically find the other two nodes x_C and y_C of the triple. Additional complications arise in ensuring that the list coloring instance of the pairs is a d1LC instance, i.e., that the size of the joint available color palette of x_C and z_C exceeds the maximum degree in the virtual graph induced by the pairs. The last difficulty that appears is solving the d1LC instance, as the bandwidth between the nodes within a pair is very limited and existing d1LC algorithms cannot be run in a black-box manner.

Further related work. Graph coloring is fundamental to distributed computing as an elegant way of breaking symmetry and avoiding contention, and was, in fact, the topic of the original paper introducing the LOCAL model [37]. There is an abundance of efficient deterministic and randomized $\Delta+1$-coloring algorithms in LOCAL and CONGEST for various settings, e.g., [6, 35, 23, 12, 5, 43, 39, 29, 32, 30, 24]. The excellent monograph on distributed graph coloring by Barenboim and Elkin is still a great resource for older results [7].

There are significantly fewer results for coloring with fewer than $\Delta+1$ colors. A LOCAL algorithm is known for $\Delta-k$-coloring in graphs not containing too large cliques [4]. An $O(\log\log n)$-round Δ-coloring algorithm in the LOCAL model is known for trees [11], matching the lower bound [9] within a constant factor. Additionally, there are works coloring special graph classes such as coloring planar graphs with 6 or 5 colors in $O(\log n)$ rounds with a deterministic LOCAL algorithm [14, 42].

Outline. In Section 2, we define the notion of slack and state required results from prior work on solving d1LC and computing an almost clique decomposition (ACD). In Section 3, we present our Δ-coloring algorithm with essentially all proofs. The algorithm consists of 5 phases and all phases except for Phases 1 (ACD computation) and Phase 2 are deterministic reductions to various d1LC instances. In Phase 2, we provide slack to sparse nodes and the nodes in ordinary cliques; this refers to part a) and part c) described in Section 1.2. For ease of presentation, the (involved) Phase 2 is presented in a top down manner. In Section 4 we present the high level overview of this phase. In essence its a reduction to solving four different subproblems. The details of the reduction are deferred to Appendix A. The heart of our approach is actually solving each of these subproblems via an instance of the constructive Lovász Local Lemma. As this part is extremely technical and cannot fit into the space constraints of a conference publication we defer this part to the full version of the paper.

2 Preliminaries: d1LC, Slack, Almost-Clique Decomposition, Graytone

In the $deg+1$-list coloring (d1LC) problem, each node of a graph receives as input a list of allowaed colors whose size exceeds its degree. The goal is to compute a proper vertex coloring in which each node outputs a color from its list. The problem can be solved with a simple centralized greedy algorithm, and it also admits efficient distributed algorithms.

▶ **Lemma 2** (List coloring [30, 34])**.** *There is a randomized* CONGEST *algorithm to* $(deg+1)$*-list-color (d1LC) any graph in* $O(\log^5 \log n)$ *rounds, w.h.p. This reduces to* $O(\log^3 \log n)$ *rounds when the degrees and the size of the color space is* $\mathrm{poly}(\log n)$*.*

The slack of a node (potentially in a subgraph) is defined as the difference between the size of its palette and the number of uncolored neighbors (in the subgraph).

▶ **Definition 3** (Slack)**.** *Let v be a node with color palette $\Psi(v)$ in a subgraph H of G. The slack of v in H is the difference $|\Psi(v)|-d$, where d is the number of uncolored neighbors of v in H.*

We use the following helpful terminology.

▶ **Definition 4** (Graytone [22])**.** *Consider an arbitrary step of the algorithm. A node is* gray *if it has unit-slack or a neighbor that will be colored in a later step of the algorithm. A node is* grayish *if it is not gray but has a gray neighbor. A set of gray and grayish nodes is said to be* graytone*.*

Any graytone set can be colored as two d1LC instances: first the grayish nodes and then the gray. We emphasize that the graytone property depends on the order in which nodes are processed. It always refers to a certain step of the algorithm in which we color the respective set. Throughout our algorithm we aim at making more and more nodes graytone.

A proof of the following construction central to our approach is given in Appendix B.

▶ **Lemma 5** (ACD computation [1, 22]). *For any graph $G = (V, E)$, there is a partition (*almost-clique decomposition (ACD) *of V into sets V_{sparse} and $C_1, C_2, \ldots, C_t$ such that each node in V_{sparse} is $\Omega(\epsilon^2\Delta)$-sparse and for every $i \in [t]$,*

(i) $(1 - \varepsilon/4)\Delta \leq |C_i| \leq (1 + \varepsilon)\Delta$,

(ii) *Each $v \in C_i$ has at least $(1 - \varepsilon)\Delta$ neighbors in C_i: $|N(v) \cap C_i| \geq (1 - \varepsilon)\Delta$,*

(iii) *Each node $u \notin C_i$ has at most $(1 - \varepsilon/2)\Delta$ neighbors in C_i: $|N(u) \cap C_i| \leq (1 - \varepsilon/2)\Delta$.*

Further, there is an $O(1)$-round CONGEST *algorithm to compute a valid ACD, w.h.p.*

We say that nodes in V_{sparse} are *sparse* and other nodes are *dense*. It is immediate from Lemma 5 that each dense node has external degree (or neighbors outside its AC) at most $\varepsilon\Delta$ and at most $2\varepsilon\Delta$ non-neighbors in its AC. Also, any pair of nodes in C_i have at least $(1 - 3\varepsilon)\Delta \geq 3\Delta/4$ common neighbors in C_i.

Notation. For a graph $G = (V, E)$ and two nodes $u, v \in V$, let $\text{dist}_G(u, v)$ denote the length of a shortest (unweighted) path between u and v in G. For a set $S \subseteq V$ we denote $\text{dist}_G(v, S) = \min_{u \in S} \text{dist}_G(v, u)$. $N(v)$ denotes the set of neighbors of a node $v \in V$.

3 Δ-Coloring in CONGEST

In this subsection, we prove the following theorem.

▶ **Theorem 1.** *There is a randomized* $\text{poly}\log\log n$*-round* CONGEST *algorithm to Δ-color any graph with maximum degree $\Delta \geq 3$. The algorithm works with high probability.*

The extreme cases of very large Δ and very small Δ can be solved in the claimed runtime with prior work [22, 40], see the proof of Theorem 1 in Section 3.3. Here, we present an algorithm for the most challenging regime where $\Delta \in O(\text{poly}\log n) \cap \Omega(\text{poly}\log\log n)$.

In the extreme case that $\Delta = \omega(\log^{21} n)$, the Δ-coloring algorithm from [22] even runs in $O(\log^* n)$ rounds. A lower bound of $\Omega(\log_\Delta \log n)$ rounds in the LOCAL model for the Δ-coloring problem [9] rules out a $O(\log^* n)$ algorithm for small Δ. Hence, in this section, we aim for an algorithm using $\text{poly}\log\log n$ rounds. In fact, we reduce the Δ-coloring problem to a few list coloring instances and a few LLL instances, each of which we solve in $\text{poly}\log\log n$ rounds.

3.1 Fine-Grained ACD Partition

The following definitions of types of almost-cliques are crucial for all results of the paper. The reader is hereby warned to read them slowly!

▶ **Definition 6** (Types of almost-cliques). *For an AC C, let $e_C = \Delta - |C| + 1$. An AC is* easy *if it contains a non-edge or a node of degree less than Δ. A node $v \notin C$ is an* intrusive *neighbor of a non-easy C if v has at least $2e_C$ neighbors in C. A non-easy AC is* difficult *if it has an intrusive neighbor. Each difficult AC C arbitrarily selects one of its intrusive neighbors as its* special *node s_C. An AC is* nice *if it is easy or if it is both non-difficult and contains a special node (necessarily for another AC). An AC is* ordinary *if it is neither nice nor difficult.*

Note that all ACs except the easy are proper cliques and all nodes in such a clique C have external degree e_C. We say that a node is ordinary (difficult, nice) if it belongs to an ordinary (difficult, nice) AC, respectively. The difficult ACs are divided into *levels*.

▶ **Definition 7** (Levels of difficult ACs). *The* maximum level ∞ *contains all difficult ACs whose special node is not contained in a difficult AC. A difficult AC C that is not at the maximum level has* level $\ell(C) = \lceil \log_2 e_C \rceil$.

Observe that $\ell(C) \leq \log_2 \Delta = O(\log \log n)$ for all difficult ACs.

▶ **Definition 8** (Node classification). *The nodes are partitioned into the following sets:*

1. *$\mathcal{S}$: the set of special nodes that are not in difficult ACs,*
2. *$\mathcal{D}_\ell$: nodes in difficult ACs of level ℓ, $\ell \in [\lg \Delta] \cup \{\infty\}$ (might include special nodes),*
3. *$\mathcal{N}$: nodes in nice ACs, excluding those in $\mathcal{S}$,*
4. *$\mathcal{O}$: nodes in ordinary ACs, and*
5. *V_*: nodes in V_{sparse}, excluding those in $\mathcal{S}$.*

Our classification is built on [22] but is subtly different and more fine-grained. We are driven by a need to limit the reach of probabilistic arguments, being that we are in the challenging sub-logarithmic degree range. Thus, a strictly smaller set of dense nodes (the ordinary) needs probabilistic slack in our formulation. On the other hand, the easy, difficult, and nice definitions are more inclusive here. The difficult ones are here divided into super-constant number of levels, as opposed to only two types in [22].

The underlying idea is to ensure that every node gets at least one unit of slack, ensuring that it can be colored as part of a d1LC instance. Easy nodes have such slack from the start; difficult ones get it from their special nodes (special nodes are used in several different ways to provide slack); sparse and ordinary nodes get it from probabilistic slack generation; and non-easy nice ones get it from same-coloring a non-edge it contains. The most challenging part of the low-degree regime is the probabilistic part. That has guided our definition, resulting in the ordinary ACs being defined as restrictively as possible and, in fact, much more restrictive than the ordinary ACs in [22].

3.2 Algorithm for Δ-coloring

Our Δ-coloring algorithm consists of the following five phases.

Algorithm 1 Δ-coloring.

```
1: Compute an ACD ($\varepsilon = 1/172$) and form the ordered partition of the nodes.
2: Color sparse nodes $V_*$ and ordinary nodes $\mathcal{O}$
3: Color nice nodes $\mathcal{N}$
4: For increasing $1 \leq \ell < \infty$ :
       Color difficult nodes $\mathcal{D}_\ell$ in level $\ell$
5: Color difficult nodes in $\mathcal{D}_\infty$ and special nodes in $\mathcal{S}$
```

The remainder of the paper describes these phases in detail. Only Phases 1 and 2 are randomized. Phase 2 is also the most involved part of our algorithm. For ease of presentation, we defer its details when Δ is at most logarithmic to Section 4. In this section, we present Phase 2 in the case of $\Delta \geq c \log n$ for a sufficiently large constant c, where Phase 2 does not require any LLL and which is sufficient to understand how Phase 2 interacts with the remaining phases. The remaining phases are identical in both cases.

3.2.1 Phase 1: Partitioning the Nodes

We first apply Lemma 5 to compute an ACD for $\varepsilon = 1/172$ and break the graph into nice ACs, difficult ACs, ordinary ACs, and the remaining nodes in V_* according to Definition 8.

3.2.2 Phase 2: Sparse and Ordinary Nodes ($\Delta \gg \log n$)

In this subsection, we prove the following lemma.

▶ **Lemma 9.** *There exists a* $\operatorname{poly}\log\log n$*-round* CONGEST *algorithm that w.h.p. colors the sparse nodes and nodes in ordinary cliques if* $\Delta \geq c \log n$ *for a sufficiently large constant* c*.*

Lemma 9 essentially follows from the proof of Lemma 3.5 in [22, arxiv version]. However, as we have changed the definition of ordinary cliques, we spell out the required details.

Slack generation is based on trying a random color for a subset of nodes. Sample a set of nodes and a random color for each of the sampled nodes. Nodes keep the random color if none of their neighbors choose the same color. See Algorithm 3 for a pseudocode. If there are enough non-edges in a node's neighborhood, then it probabilistically gets significant slack.

Algorithm 2 Phase 2: Coloring Sparse and Ordinary Nodes (when $\Delta \gg \log n$).

```
Run SlackGeneration on $V_* \cup \mathcal{O}$
Color the remaining ordinary nodes $\mathcal{O}$
Color the remaining sparse nodes $V_*$
```

Algorithm 3 SlackGeneration.

```
Input: $S \subseteq V$
Each node in $v \in S$ is active w.p. $1/20$
Each active node $v$ samples a color $r_v$ u.a.r. from $[\chi]$.
$v$ keeps the color $r_v$ if no neighbor tried the same color.
```

We also require the following lemma from [22].

▶ **Lemma 10** ([22])**.** *Let* C *be a non-easy AC,* $S \subseteq V$ *be a subset of nodes containing* C*, and* M *be an arbitrary matching between* C *and* $N(C) \setminus C$*. Then, after* SlackGeneration *is run on* S*,* C *contains* $\Omega(|M|)$ *uncolored nodes with unit-slack in* $G[S]$*, with probability* $1 - \exp(-\Omega(|M|))$*.*

There exists a large matching satisfying the hypothesis of Lemma 10,

▶ **Lemma 11.** *For each ordinary AC* C*, there exists a matching* M_C *between* C *and* $N(C) \setminus C$ *of size* $2\Delta/5$*.*

Proof. We use the following combinatorial result that is proven in Appendix B for completeness.

▷ Claim 12. Let $B = (Y, U, E_B)$ be a bipartite graph where nodes in Y have degree at least k and nodes in U have degree at most $2k$. There exists a matching of size $|Y|/2$ in B.

Proof. Let M be a maximum matching in B and suppose that more than half the nodes in Y are unmatched. Let S be the set of nodes reachable from the unmatched nodes $Y \setminus V(M)$. Since M has no augmenting path, S contains no unmatched node of U. All of the $|Y \cap S| \cdot k$

edges incident on $Y \cap S$ have their other endpoint in $U \cap S$. By the degree bound on U, there are fewer than $|U \cap S|2k$ such edges. Thus, $|Y \cap S| < 2|U \cap S|$. Every node in $U \cap S$ is matched to a node in $Y \cap S$, while all unmatched nodes in Y are in $Y \cap S$. Thus, the number of unmatched nodes in Y is at most $|Y \cap S| - |U \cap S| < |U \cap S| \leq |M|$. This is a contradiction, and hence, at least half the nodes in Y are matched. ◁

As C is not easy, all its nodes have external degree e_C, while nodes in $N(C) \setminus C$ are by assumption not intrusive neighbors of C, so they have at most $2e_C$ neighbors in C. Claim 12 then implies that there exists a matching between C and $N(C) \setminus C$ of size $|C|/2 \geq (1-\epsilon)\Delta/2 \geq 2\Delta/5$. ◀

The properties of Phase 2 are summarized in the following lemma.

▶ **Lemma 13.** *If $\Delta \geq c \log n$ for a sufficiently large constant c, the following properties hold w.h.p. after Step 1 of Algorithm 2:*
(†) *Each sparse node has unit-slack in $G[V_*]$,*
(††) *Each ordinary AC has an uncolored unit-slack node in $G[V_* \cup \mathcal{O}]$.*

Proof. We run SLACKGENERATION on the node set $S = V^* \cup \mathcal{O}$. Nodes with neighbors outside $V^* \cup \mathcal{O}$ have slack while the rest of the graph is stalled. We focus on the remaining nodes. Each sparse node gets the respective slack with probability at least $1 - \exp(\Omega(\Delta))$ [18, Lemma 3.1], implying (†). By Lemma 11, there is a matching between C and $N(C) \setminus C$ of size $2\Delta/5$. Thus, (††) holds with probability at least $1 - \exp(-\Omega(\Delta))$, by Lemma 10.

Both probabilities become w.h.p. guarantees if $\Delta \geq c \log n$ for a sufficiently large constant c. For $\Delta \geq \Delta_0$ for a sufficiently large constant Δ_0 we obtain an LLL. ◀

Proof of Lemma 9. By Lemma 13 w.h.p. all sparse nodes become gray as they have unit slack. Also, the unit-slack node in each ordinary AC becomes gray and all other nodes of the AC become grayish as ordinary ACs induce cliques. This is sufficient to color all nodes with $O(1)$ d1LC instances. ◀

Forward pointer. The main difficulty of Phase 2 for smaller values of Δ is to mimic the properties of Lemma 13. Section 4 are devoted to ensuring these properties via several LLLs and d1LC instances that can be solved in a bandwidth-efficient manner.

3.2.3 Phase 3: Nice ACs

We give a simpler treatment than [22]. We want a *toehold* in each nice AC: a node with permanent or temporary slack. With a toehold, the rest is easy. Namely, ACs have all nodes of internal degree at least $(1-\varepsilon)\Delta$, of which none are colored in previous phases. The neighbors of a toehold are gray, and there are at least $(1-\varepsilon/4)\Delta$ of them by Lemma 5, all uncolored. The remaining nodes in the AC are then grayish, so the AC is graytone.

Nice ACs come in three types, depending on if they contain a special node, a non-edge, or a degree-below-Δ node. The first and third types immediately give us a toehold. It remains then to consider nice ACs with a non-edge but with no special node, which we call *hollow*.

For a hollow AC C, we identify an arbitrary non-edge (u_C, w_C) and call it *the pair for* C. We color the pairs for hollow ACs as a d1LC instance. The two nodes in a pair have at least $\Delta/2$ common neighbors within C and any of them can function as a toehold. It remains to argue that we can find a valid coloring of the pairs efficiently.

▶ **Lemma 14.** *The pairs of hollow ACs can be colored in the* CONGEST *model in* $O(\log^3 \log n)$ *rounds.*

Proof. As the nodes of a hollow C were uncolored, the only nodes that can conflict with the coloring of the pair are the at most $2 \cdot \varepsilon\Delta \leq \Delta/2$ external neighbors. The $\Delta + 1$ colors we have to work with significantly exceed that. Thus, the pairs are $deg + 1$-list colorable.

Both nodes of the pair (u_C, w_C) have at least $(1 - \epsilon)\Delta$ neighbors in C, so they have at least $(1 - \epsilon)\Delta - (|C| - (1 - \epsilon)\Delta) > (1 - 3\epsilon)\Delta \geq \Delta/2$ common neighbors in C. They provide the bandwidth to transmit to one node all the colors adjacent to the other node. Also, all messages to and from u_C *vis-a-vis* its external neighbors can be forwarded in two rounds. Hence, we can simulate any CONGEST coloring algorithm on the pairs with $O(1)$-factor slowdown; in particular, we can simulate the algorithm from Lemma 2. ◀

3.2.4 Phase 4: Difficult ACs in a Non-Maximum Level

By Definition 7, the special node s_C of any difficult AC C at a level other than D_∞ is contained in another difficult AC $C' \neq C$. The next lemma shows that the level of C' must be strictly larger than the level of C, which allows us to color C fast while C' remains uncolored.

▷ Claim 15. For an AC C with $\ell(C) < \infty$, let C' be the difficult AC that contains the special node s_C. Then we have $\ell(C) < \ell(C')$.

Proof. The special node s_C has external degree of at least $2e_C$ as it is connected to at least $2e_C$ nodes of C that do not lie within C'. Hence, we obtain that the external degree $e_{C'}$ in AC C' is at least $e_{C'} \geq 2e_C$, so $\ell(C') > \ell(C)$. ◁

We color all ACs of a level in parallel, in increasing order of levels. Due to the previous claim, the special node of an AC is contained in a difficult clique in a larger level or not contained in a difficult clique at all. Hence, the special node is uncolored when the clique is processed. So, when processing some level $1 \leq i \leq O(\log \log n)$, we color all nodes in ACs of that level, but we do not color their respective special nodes. Thus, the respective special node provides a toehold for the respective clique.

3.2.5 Phase 5: Difficult ACs in the Maximum Level

The maximum level is processed last and differently from the other levels. By definition, the special node s_C of an AC in ∞ level is not contained in a difficult AC. Also, all nodes in D_∞ and their special nodes are still uncolored at the beginning of this phase.

The algorithm has four steps: (1) Form pairs of selected non-adjacent nodes, (2) Color the nodes in each pair consistently, (3) Graytone color the remaining nodes of the AC, and (4) Color the special nodes $\mathcal{S}$. We explain each step in detail.

First, we form the following pairs. For each special node s_C that is special for only one AC C at level ∞: Form a *type-1* pair $T_s = (s_C, u_C)$ with a non-neighbor of s_C in C. For each special node s that is special for more than one ACs at level ∞, form a *type-2* pair $T_s = (w_1, w_2)$, where w_1 and w_2 are arbitrary non-adjacent nodes in two of the ACs for which s is special. Let $\mathcal{E}$ be the set of the latter special nodes.

▷ Claim 16. The pairs can be properly formed.

Proof. *Type-1:* An (uncolored) non-neighbor u_C of p_C exists as p_C can have at most $(1-\varepsilon/2)\Delta$ neighbors in C by Lemma 5 (4), but the AC C has at least $(1 - \varepsilon/4)\Delta$ vertices.

Type-2: Let C_1 and C_2 be two ACs at level ∞ for which s is special, where $e(C_1) \leq e(C_2)$. By definition, s has at least $2e(C_1)$ $(2e(C_2))$ neighbors in C_1 (C_2), respectively. Pick w_1 to be any neighbor of s in C_1. Node w_1 has at most $e(C_1)$ neighbors in C_1. Thus, there are at least $2e(C_2) - e(C_1) > 0$ nodes in C_2 that are neighbors of s and non-neighbors of w_1, and we can pick any such node as w_2. ◁

▶ **Lemma 17.** *Coloring the pairs is a* $(deg+1)$*-list coloring instance that can be solved in* $\operatorname{poly}\log\log n$ *rounds in* CONGEST*, w.h.p.*

Proof. *Type-1 pair* $T = \{s_C, u_C\}$*,* $s_C \notin C$*,* $u_C \in C$*:* We say that a node *conflicts* with the pair $\{s_C, u_C\}$ if the node is already colored or is contained in an adjacent pair of the same phase. As C does not contain a special node, u_C is the only node of C participating in the phase and all other nodes of C are still uncolored. The node u_C can only be adjacent to e_C conflicting nodes as it has external degree at most e_C. As s_C has at least $2e_C$ neighbors in C, it can conflict with at most $\Delta - 2e_C$ nodes. Thus, the pair conflicts with at most $e_C + \Delta - 2e_C = \Delta - e_C$ nodes, which is less than Δ, the number of colors initially available. Thus, the problem of coloring such pairs is a $(deg+1)$-list coloring problem.

Type-2 pair $T = \{w_1, w_2\}$*:* Each such pair (w_1, w_2) is adjacent to at most $e(C_1)+e(C_2) \leq 2\epsilon\Delta$ nodes in other ACs. Further, all nodes in the ACs C_1 and C_2 are still uncolored, so both nodes have at least $(1-2\epsilon)\Delta$ colors in their palette, and each pair is adjacent to at most $2\varepsilon\Delta$ other pairs or already colored neighbors, that is, the palette exceeds the degree.

CONGEST *Implementation.* A type-1 pair has at least $e(C)$ common neighbors (the special node s_C has $2e(C)$ neighbors inside the clique by its definition that are all connected to u_C), which suffices to communicate the colors and all messages of external neighbors of u_C to s_C (u_C has at most e_C external neighbors). Hence, the coloring can be achieved in CONGEST.

Let s be the common special node of a type-2 pair $\{w_1, w_2\}$ and let C_1 and C_2 be the respective cliques. For $i = 1, 2$ the node w_i has at most e_{C_i} outside neighbors and s has $2e_{C_i} \geq e_{C_i}$ neighbors in C_i, denote these by X_i. We simulate the pair by s. The node w_i can forward all initial colors of outside neighbors as well as all messages from them to s by relaying them through X_i. ◀

After coloring the pairs, each difficult AC C has a node with unit-slack in $G[V \setminus \mathcal{E}]$, either because the clique contains an uncolored node with two neighbors appearing in a consistently colored type-1 pair $T = \{s_C, u_C\}$, or because it contains an uncolored node with a neighbor in $\mathcal{E}$. In the former case, the uncolored node exists because s_C has at least one neighbor in C that is also a neighbor of u_C. In the latter case, the special node s with type-2 pair $T = \{w_1, w_2\}$ has by definition further neighbors besides w_1 and w_2 in each clique that are all uncolored.

Thus, we color all nodes in difficult cliques via the graytone property. At the end, we color the nodes in $\mathcal{E}$, which have unit-slack as they are adjacent to a type-2 pair.

3.3 Proof of Theorem 1

Proof of Theorem 1. There are five cases, depending on the relation of Δ and n. Generally, we use Lemma 2 to solve d1LC instances in $\operatorname{poly}\log\log n$ rounds. Whenever the d1LC instances require additional arguments to be solved in the respective time, e.g., because they are defined on a virtual graph, we reason their runtime when they are introduced.

- If $\Delta = \omega(\log^4 n)$, we use the algorithm from [22] to Δ-color the graph.
- For $c\log n \leq \Delta = O(\log^4 n)$ for a sufficiently large constant c, the result follows by executing Algorithm 1 with the arguments of this section. Phases 1–3 only require $O(1)$ rounds and a constant number of d1LC instances. In Phase 4, we iterate through the $O(\log\Delta) = O(\log\log n)$ levels and solve a constant number of d1LC instances for each level. Phase 5 can be executed in $\operatorname{poly}\log\log n$ time by Lemma 17.

- When $\operatorname{poly}\log\log n \leq \Delta \leq c\log n$, we use Algorithm 1 from this section and replace Phase 2 with Algorithm 4 (presented in Section 4) whose correctness and runtime we prove in Section 4.
- If $\Delta_0 \leq \Delta \leq \operatorname{poly}\log\log n$, we use the algorithm of this section together with the LLL representation from the proof of Lemma 13. The LLL can be solved with the CONGEST LLL solver of [40] in $\operatorname{poly}\Delta \operatorname{poly}\log\log n = \operatorname{poly}\log\log n$ rounds. Here, Δ_0 is a sufficiently large constant such that the LLL guarantees from Lemma 13 hold.
- If $3 \leq \Delta \leq \Delta_0$, that is, for constant Δ, there is an existing algorithm from [40].

In all cases, the algorithm runs in $\operatorname{poly}\log\log n$ rounds. ◀

4 Phase 2 ($\Delta = O(\log n)$): Sparse Nodes and Ordinary Cliques

In this section, we deal with Phase 2 for the most challenging regime of $\Delta \in O(\log n) \cap \Omega(\operatorname{poly}\log\log n)$. The following lemma follows from all proofs in this section, together with Lemmas 20, 22, and 23 stated in Appendix A and proven in the full version of this paper.

▶ **Lemma 18** (Phase 2). *There exists a* $\operatorname{poly}\log\log n$*-round* CONGEST *algorithm that w.h.p. color the sparse nodes and nodes in ordinary cliques if* $\log^{10}\log n \leq \Delta \leq O(\log n)$.

We first give high-level ideas of our method. We divide the ordinary cliques into the *small*, of size at most $\Delta(1 - 1/(10\log^3\log n))$, and the *large*. Nodes in small ordinary cliques have significant sparsity (i.e., non-edges in their induced neighborhood), which means that the one-round procedure of trying a random color has a good probability of successfully generating slack. The natural LLL formulation of that step is therefore well-behaved enough that it can be solved fast in CONGEST with a few additional tweaks.Large nodes need a different approach.

For each large AC, we produce unit slack for a single node. See Figure 3 for an illustration of the process we will describe. We identify for each such AC a triplet of nodes (x, y, z) with the objective to color x and z with the same color, while y remains uncolored. This way, y receives unit slack, which gives us a toehold to color the whole AC.

Computing such triplets is non-trivial. We do so by breaking it into three steps, each solvable by a different LLL formulation. In brief, we first compute a set Z of candidate z-nodes; next partition Z into two sets; and then select the actual z-nodes to be used from these two sets. The split of Z into two sets is required to make the process of finally finding the z-nodes fit the LLL solver from [33]. The properties of the set Z imply that it is then much easier to identify compatible x- and y-nodes, and once we find such triplets, we set up a virtual coloring instance for same-coloring x- and z-nodes in each triple. We show that this instance is d1LC and can be solved with low bandwidth despite being defined on a virtual graph. This provides a slack-toehold to the y-node of each triple and the coloring can be extended via d1LC instances to the whole instance.

Algorithm. The first step of the algorithm is to compute a large matching M_C between each ordinary clique C and $N(C) \setminus C$ in parallel. We then classify the ordinary cliques as follows. Fix the parameter $q(n) = 10\log^3\log n$ throughout this section.

▶ **Definition 19** (Small, Large, Unimportant and Important Ordinary cliques.). *An ordinary AC is* large *if it contains more than* $\Delta - \Delta/q(n)$ *nodes, and* small *otherwise. A large AC is* important *if* $|(V(M_C) \setminus C) \cap \mathcal{O}_l| \geq \Delta/12$*, and* unimportant *otherwise.*

We say that a node is small/large/important/unimportant if it belongs to an AC of the corresponding type. Let $\mathcal{O}_i$, $\mathcal{O}_u$, $\mathcal{O}_l = \mathcal{O}_i \cup \mathcal{O}_u$, and $\mathcal{O}_s$ be the set of important, unimportant, large, and small nodes, respectively.

Next, we summarize the high level steps of the algorithm.

Algorithm 4 Phase 2: Coloring Sparse and Ordinary Nodes ($\Delta = O(\log n)$).

```
1: Step 0: For each ordinary AC $C$ in parallel, compute a matching $M_C \subseteq C \times (N(C) \setminus C)$.
   Classify ordinary ACs into important, unimportant, and small ACs.
2: Step 1: Generate slack for sparse and small nodes (via LLL, see full version)
3: Step 2: Compute candidate sets $Z = Z_1 \cup Z_2 \subseteq \mathcal{O}_l$ (via LLL, see full version)
4: Step 3: Form triples $(x_C, y_C, z_C) \in C \times C \times Z$ (via LLL, see full version)
5: Step 4: Same-color $(x, z)$-pairs via virtual coloring instance
6: Step 5: Color the remainder of $V^* \cup \mathcal{O}$ (via d1LC instances).
```

Steps 0,4, and 5 are detailed in Appendix A. While Steps 1–3 are the heart of this paper, their treatment is extremely technical and requires setting up several involved LLLs that are then solved with the LLL solvers of [33]. Thus, the complete treatment of these steps is defered to the full version of the paper and Appendix A only focuses on presenting the guarantees provided by these steps (Lemmas 20, 22, and 23).

References

1 Sepehr Assadi, Pankaj Kumar, and Parth Mittal. Brooks' theorem in graph streams: a single-pass semi-streaming algorithm for Δ-coloring. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 234–247, 2022.

2 Alkida Balliu, Sebastian Brandt, Fabian Kuhn, and Dennis Olivetti. Distributed Δ-coloring plays hide-and-seek. In *Proc. 54th ACM Symp. on Theory of Computing (STOC)*, 2022.

3 Alkida Balliu, Keren Censor-Hillel, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Locally checkable labelings with small messages. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.DISC.2021.8.

4 Étienne Bamas and Louis Esperet. Distributed coloring of graphs with an optimal number of colors. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 126 of *LIPIcs*, pages 10:1–10:15. LZI, 2019. doi:10.4230/LIPICS.STACS.2019.10.

5 Philipp Bamberger, Fabian Kuhn, and Yannic Maus. Efficient deterministic distributed coloring with small bandwidth. In *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 243–252, 2020. doi:10.1145/3382734.3404504.

6 L. Barenboim. Deterministic (Δ + 1)-coloring in sublinear (in Δ) time in static, dynamic and faulty networks. In *Proc. 34th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 345–354, 2015.

7 Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments.* Morgan & Claypool Publishers, 2013. doi:10.2200/S00520ED1V01Y201307DCT011.

8 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM*, 63(3):20:1–20:45, 2016. doi:10.1145/2903137.

9 Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed Lovász local lemma. In *Proc. 48th ACM Symposium on Theory of Computing (STOC 2016)*, pages 479–488. ACM, 2016. doi:10.1145/2897518.2897570.

10 R. Leonard Brooks. On colouring the nodes of a network. *Mathematical Proceedings of the Cambridge Philosophical Society*, 37(2):194–197, 1941. doi:10.1017/S030500410002168X.

11 Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. Distributed edge coloring and a special case of the constructive Lovász local lemma. *ACM Trans. Algorithms*, 2020. doi:10.1145/3365004.

12 Yi-Jun Chang, Wenzheng Li, and Seth Pettie. An optimal distributed (Δ+1)-coloring algorithm? In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 445–456, 2018. doi:10.1145/3188745.3188964.

13 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. *SIAM J. Comput.*, 48(1):33–69, 2019. doi:10.1137/17M1157957.

14 Shiri Chechik and Doron Mukhtar. Optimal distributed coloring algorithms for planar graphs in the LOCAL model. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 787–804. SIAM, 2019. doi:10.1137/1.9781611975482.49.

15 Kai-Min Chung, Seth Pettie, and Hsin-Hao Su. Distributed algorithms for the Lovász local lemma and graph coloring. *Distributed Comput.*, 30(4):261–280, 2017. doi:10.1007/S00446-016-0287-6.

16 Sam Coy, Artur Czumaj, Peter Davies, and Gopinath Mishra. Parallel derandomization for coloring, 2024. Note: https://arxiv.org/abs/2302.04378v1 contains the Delta-coloring algorithm. arXiv:2302.04378.

17 Peter Davies. Improved distributed algorithms for the Lovász local lemma and edge coloring. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 4273–4295. SIAM, 2023. doi:10.1137/1.9781611977554.CH163.

18 Michael Elkin, Seth Pettie, and Hsin-Hao Su. ($2\Delta - 1$)-edge-coloring is much easier than maximal matching in the distributed setting. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 355–370, 2015. doi:10.1137/1.9781611973730.26.

19 Paul Erdös and László Lovász. Problems and Results on 3-chromatic Hypergraphs and some Related Questions. *Colloquia Mathematica Societatis János Bolyai*, pages 609–627, 1974.

20 Manuela Fischer. Improved deterministic distributed matching via rounding. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 17:1–17:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.17.

21 Manuela Fischer and Mohsen Ghaffari. Sublogarithmic Distributed Algorithms for Lovász Local Lemma, and the Complexity Hierarchy. In *the Proceedings of the 31st International Symposium on Distributed Computing (DISC)*, pages 18:1–18:16, 2017. doi:10.4230/LIPIcs.DISC.2017.18.

22 Manuela Fischer, Magnús M. Halldórsson, and Yannic Maus. Fast distributed Brooks' theorem. In *Proceedings of the SIAM-ACM Symposium on Discrete Algorithms (SODA)*, pages 2567–2588, 2023. doi:10.1137/1.9781611977554.ch98.

23 Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 625–634, 2016. doi:10.1109/FOCS.2016.73.

24 Marc Fuchs and Fabian Kuhn. List defective colorings: Distributed algorithms and applications. In Rotem Oshman, editor, *37th International Symposium on Distributed Computing, DISC 2023, October 10-12, 2023, L'Aquila, Italy*, volume 281 of *LIPIcs*, pages 22:1–22:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.DISC.2023.22.

25 Mohsen Ghaffari. Distributed maximal independent set using small messages. In *Proc. 30th Symp. on Discrete Algorithms (SODA)*, pages 805–820, 2019. doi:10.1137/1.9781611975482.50.

26 Mohsen Ghaffari, David G. Harris, and Fabian Kuhn. On derandomizing local distributed algorithms. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 662–673, 2018. doi:10.1109/FOCS.2018.00069.

27 Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, and Yannic Maus. Improved distributed delta-coloring. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 427–436, 2018. URL: https://dl.acm.org/citation.cfm?id=3212764.

28 Mohsen Ghaffari and Fabian Kuhn. Deterministic distributed vertex coloring: Simpler, faster, and without network decomposition. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1009–1020, 2021. doi:10.1109/FOCS52979.2021.00101.

29 Magnús M. Halldórsson, Fabian Kuhn, Yannic Maus, and Tigran Tonoyan. Efficient randomized distributed coloring in CONGEST. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 1180–1193, 2021. Full version at CoRR abs/2105.04700. doi:10.1145/3406325.3451089.

30 Magnús M. Halldórsson, Fabian Kuhn, Alexandre Nolin, and Tigran Tonoyan. Near-optimal distributed degree+1 coloring. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 450–463. ACM, 2022. doi:10.1145/3519935.3520023.

31 Magnús M. Halldórsson, Yannic Maus, and Alexandre Nolin. Fast distributed vertex splitting with applications. In Christian Scheideler, editor, *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*, volume 246 of *LIPIcs*, pages 26:1–26:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.DISC.2022.26.

32 Magnús M. Halldórsson and Alexandre Nolin. Superfast coloring in CONGEST via efficient color sampling. In Tomasz Jurdzinski and Stefan Schmid, editors, *Structural Information and Communication Complexity - 28th International Colloquium, SIROCCO 2021, Wrocław, Poland, June 28 - July 1, 2021, Proceedings*, volume 12810 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2021. doi:10.1007/978-3-030-79527-6_5.

33 Magnús M. Halldórsson, Yannic Maus, and Saku Peltonen. Distributed Lovász local lemma under bandwidth limitations, 2024. arXiv:2405.07353, doi:10.48550/arXiv.2405.07353.

34 Magnús M. Halldórsson, Alexandre Nolin, and Tigran Tonoyan. Overcoming congestion in distributed coloring. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 26–36. ACM, 2022. doi:10.1145/3519270.3538438.

35 David G. Harris, Johannes Schneider, and Hsin-Hao Su. Distributed $(\Delta + 1)$-coloring in sublogarithmic rounds. *Journal of the ACM*, 65:19:1–19:21, 2018. doi:10.1145/3178120.

36 Öjvind Johansson. Simple distributed $\Delta + 1$-coloring of graphs. *Inf. Process. Lett.*, 70(5):229–232, 1999.

37 Nati Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

38 Yannic Maus, Saku Peltonen, and Jara Uitto. Distributed symmetry breaking on power graphs via sparsification. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, PODC '23, pages 157–167, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3583668.3594579.

39 Yannic Maus and Tigran Tonoyan. Local conflict coloring revisited: Linial for lists. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 16:1–16:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.16.

40 Yannic Maus and Jara Uitto. Efficient CONGEST algorithms for the Lovász local lemma. In Seth Gilbert, editor, *Proceedings of the International Symposium on Distributed Computing (DISC)*, volume 209 of *LIPIcs*, pages 31:1–31:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.DISC.2021.31.

41 Alessandro Panconesi and Aravind Srinivasan. The local nature of Δ-coloring and its algorithmic applications. *Combinatorica*, 15(2):255–280, 1995. doi:10.1007/BF01200759.

42 Luke Postle. Linear-time and efficient distributed algorithms for list coloring graphs on surfaces. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 929–941. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00060.

43 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 350–363, 2020.

A Details of Phase 2

Step 0: Classifying ACs and computing matchings. We compute a matching M_C for each ordinary clique C between the vertices in C and the ones in $N(C) \setminus C$. We use a 2.5-approximate algorithm of [20] running in $O(\log^2 \Delta + \log^* n) = O(\log^2 \log n)$ rounds, obtaining that $|M_C| \geq (2\Delta/5)/2.5 = \Delta/10$, using Lemma 11.

We view the edges of M_C as being directed arcs with a head in C and tail in $V \setminus C$. Each AC can determine its size and the size of $V(M_C) \cap \mathcal{O}_l$ in $O(1)$ rounds and hence the classification of Definition 19 can be computed in $O(1)$ rounds.

Step 1: Slack for sparse and small nodes. In this step, we create slack for sparse nodes and all nodes in $\mathcal{O}_s$. The key property of small nodes is that they are relatively sparse (with many non-edges in their neighborhoods), so randomly trying colors is likely to produce slack. That leads to an LLL formulation that we can make simulatable and can therefore implement in CONGEST.

The properties are summarized by the following lemma. Besides providing slack to all sparse nodes and the nodes in small ordinary ACs, it also guarantees that each neighborhood (and hence also each AC) does not have too many nodes colored and that the matching M_C of each AC does not get too many nodes colored.

▶ **Lemma 20.** *Assume that we are given a matching M_C of size at least $\Delta/10$ between C and $N(C) \setminus C$ for each ordinary AC C. There is a* $\operatorname{poly}\log\log n$*-round (LLL-based)* CONGEST *algorithm that w.h.p. colors a subset $S \subseteq V^* \cup \mathcal{O}$ and ensures that:*

1. *Each uncolored node in $V^* \cup \mathcal{O}_s$ has unit-slack in $G[V^* \cup \mathcal{O}]$.*
2. *In each of the following subsets, at most $O(\log^4 \log n \cdot \log \Delta)$ nodes are colored: $N(v)$ for each $v \in V^* \cup \mathcal{O}$ and $V(M_C)$ for each AC C .*

Step 2: Compute triple candidate set via LLL. Let $X = \mathcal{O}_l \setminus \{v \in \mathcal{O}_l : v \text{ colored in Step 1}\}$.

The goal of this step is to compute two disjoint sets Z_1, Z_2 of uncolored nodes such that each important AC has sufficiently many matching edges satisfying the following definition of usefulness.

▶ **Definition 21** (useful edge). *Given a subset $Z \subseteq X$ and important AC C, a matched arc $\overrightarrow{vu} \in M_C$ is* useful *for C if $v \in (X \setminus Z)$ and $u \in Z$. Refer to Z as the* black *nodes and $X \setminus Z$ as the* white *nodes. An edge is* white *if both endpoints are* white.

An arc $\overrightarrow{vz}$ cannot be useful for the AC containing v; only the one containing z.

Formally, Step 2 provides the following lemma. For an AC C and set Z, let $U(C, Z)$ denote the arcs of M_C with one endpoint in Z (and the other in C).

▶ **Lemma 22.** *Let $q = 1/30$. There is a* poly log log n*-round (LLL-based)* CONGEST *algorithm computing disjoint subsets $Z_1, Z_2 \subseteq \mathcal{O}_l$ satisfying the following properties, w.h.p.:*

1. $|U(C, Z_i)| \geq q^2(1-q)^3\Delta/60$*, for $i = 1, 2$ and for each important AC C, and*
2. $|(Z_1 \cup Z_2) \cap N(v)| \leq \Delta/10$*, for all $v \in \mathcal{O}$.*

Step 3: Forming triples via LLL. The goal of this step is to compute a triple $(x_C, y_C, z_C) \in C \times C \times Z$ of nodes that satisfy the conditions of the next lemma. These triple nodes are distinct for different ACs.

▶ **Lemma 23.** *Given sets $Z_1, Z_2 \subseteq \mathcal{O}_l$ with the properties as in Lemma 22, there is a* poly log log n*-round (LLL-based)* CONGEST *algorithm that computes for each large important AC C a triple (x_C, y_C, z_C) of uncolored nodes such that w.h.p.:*

1. $x_C, y_C \in C$ *and* $z_C \notin C$*,*
2. $y_C x_C$*,* $y_C z_C \in E$*,* $x_C z_C \notin E$ *(x_C and z_C are non-adjacent; y_c is adjacent to both x_C and z_C) and*
3. *the graph induced by $\{z_C : C$ is important$\}$ has maximum degree $\leq \Delta/10$.*

We model the problem of selecting z_C for each important AC C as a disjoint variable set LLL.

Step 4: Same-coloring (x_C, z_C) pairs. Given a triple (x_C, y_C, z_C), we will create a toehold for the AC C at y_C by coloring its non-adjacent neighbors x_C and z_C with the same color.

Let H_P (P for pair) be the virtual graph consisting of one vertex for each pair (s_C, z_C) and an edge between two pairs (s_C, z_C) and $(s_{C'}, z_{C'})$ if there is any edge in G between $\{s_C, z_C\}$ and $\{s_{C'}, z_{C'}\}$. The *list of available colors* $L((s_C, z_C))$ consists of all colors that are not used by the already colored neighbors in G of s_C and z_C.

▶ **Lemma 24.** *The maximum degree Δ_{H_P} of H_P is upper bounded by $\Delta/9$.*

Proof. By Lemma 23, each node has at most $\Delta/10$ neighbors in Z. Define the set $X' = \{x_C : C$ is an important AC$\}$. As X' contains at most one node per AC, the number of neighbors that a node in $\mathcal{O}_l$ can have in U is upper bounded by its external degree plus 1, which is upper bounded by $\Delta/q(n) + 1$. Thus, the maximum degree Δ_{H_P} of the virtual graph H_P is at most $\Delta/10 + \Delta/q(n) + 1 \leq \Delta/9$ for sufficiently large n. ◀

▶ **Lemma 25.** *Coloring H_P – i.e., same-coloring the pairs – is a deg+1-list coloring instance.*

Proof. By Lemma 24 we obtain $\Delta_{H_P} \leq \Delta/9$. As we colored at most $x = O(\log^5 \log n)$ vertices in each neighborhood in Step 1, the list of available colors of each pair has at least $\Delta - 2x \gg \Delta/9 = \Delta_{H_P}$ colors available in their joint list. Hence, we obtain a *deg* + 1-list coloring instance. ◀

CONGEST implementation. Our algorithm is based on the *deg* + 1-list coloring algorithm from [25, 8]. Before we show how to color the nodes in H_P, we need to define a slow (it takes $O(\log n)$ rounds) randomized algorithm. The algorithm is used in our analysis and it works as follows. In each iteration, each uncolored pair executes the following procedure that may result in the pair to try to get colored with a color or to not try a color (also see Algorithm 5 for pseudocode of the algorithm). Throughout the algorithm, nodes x_C and z_C maintain lists $L(x_C)$ and $L(z_C)$ consisting of all colors not used by their respective neighbors in G. Then, in one iteration node x_C selects a color c u.a.r. from its list of available colors $L(x_C)$, and sends it to the other endpoint through node y_C. The other endpoint z_C checks whether

$c \in L(z_C)$; if so, both nodes agree on trying color c, and the color is sent to their neighbors. If no incident pair tries the same color, the pair gets permanently colored with the color. Lastly, both nodes individually update their lists by removing colors from adjacent vertices that got colored from their respective list. There is no explicit coordination between the two vertices in maintaining a joint list of available colors.

Algorithm 5 Randomized Pair Coloring.

```
1: Each node $x_C$ selects a color $c$ u.a.r. from $L(x_C)$ and sends $c$ to $z_C$
2: If $c \in L(z_C)$ then TryColor(c)
3: Update lists $L(x_C) \leftarrow L(x_C) \setminus \{c(v) : v \in N_G(x_C)\}$ and $L(z_C) \leftarrow L(z_C) \setminus \{c(v) : v \in N_G(z_C)\}$
```

The next lemma shows that each pair gets colored with constant probability.

▶ **Lemma 26.** *Consider an arbitrary iteration of Algorithm 5 and an arbitrary pair (x_C, z_C) for a hiding AC C that is uncolored at the start of the iteration. Then, we have*

$$\Pr((x_C, z_C) \text{ gets colored in the iteration}) \geq 1/2\ . \tag{1}$$

The bound on the probability holds regardless of the outcome of previous iterations.

Proof. Note [2] that throughout the execution of Algorithm 5 the respective lists of nodes x_C and z_C are always of size at least $\Delta - \Delta_{H_P} - \Omega(\log^5 \log n) \geq 4\Delta/5$ as $\Delta = \omega(\log^5 \log n)$ and $\Delta_{H_P} \leq \Delta/9$, by Lemma 24. Note, that both nodes keep their individual list of available colors in which they only remove the colors of immediate neighbors in G from the list of available colors. Thus, at all times we have $|L(x_C)| \cap L(z_C)| \geq 3\Delta/5$. Let X be the set of colors tried by one of the $\Delta_{H_P} \leq \Delta/9$ pairs incident to (s_C, z_C) in the current iteration. We obtain $|(L(s_C) \cap L(z_C)) \setminus X| \geq \Delta/2$. As these colors are at least half of $L(x_C)$'s palette, the probability that the pair (x_C, z_C) gets colored is at least $1/2$. ◀

▶ **Lemma 27.** *There is a randomized* poly log log n*-round* CONGEST *algorithm that w.h.p. colors the pairs of H_P.*

Proof. Consider the well-understood color trial algorithm in which nodes repeatedly try a color from their list of available colors, keep their color permanently if no neighbor tries the same color, and remove colors of permanently colored neighbors from their list of available colors. It is known that this algorithm colors each node with a constant probability in each iteration [8, 36]. Thus, it requires $O(\log n)$ rounds to color all vertices of a graph. The shattering-based CONGEST algorithm from [25] for d1LC runs in poly log log n rounds. It requires three subroutines: a) A color trial algorithm like the one from [8, 36], b) a network decomposition algorithm that can run on small subgraphs (the ones in [43, 40, 38] do the job), and c) the possibility to run $O(\log n)$ instances of the color trial algorithm in parallel. In our setting we want to solve the same problem, but on the virtual graph H_p while the communication network is still the original graph G. The subroutine for part b) can be taken from prior work as the same issue is dealt with formally in [40, 38, 33]. We refer to these works for the details and also the definition of a network decomposition. Let us sketch the main ingredient for the informed reader. Instead of computing a network decomposition of small subgraphs of H_P, the subgraphs are first projected to G, and a network decomposition

[2] The constants in this proof are not chosen optimally in order to improve readability.

of G is computed afterwards. This only requires an increased distance between clusters such that the preimage of the decomposition induces a proper network decomposition of H_P.

For ingredients a) and c), we observe that Ghaffari's algorithm only requires the following properties for the color trial algorithm: i) one iteration can be executed in constant time and with $\text{poly} \log \log n$ bandwidth, allowing to execute $O(\log n)$ instances in parallel in the CONGEST model, and ii) each node gets colored with a constant probability in each iteration. Thus, we can replace the color trial algorithm with the color trial algorithm for H_P given in Algorithm 5. We have already argued that it can be implemented with $\text{poly} \log \log n$ bandwidth showing i) and Lemma 26 provides its constant success probability for ii). ◀

Step 5: Completing the coloring. To finish the coloring, we first color the unimportant nodes and then the important, small, and sparse nodes.

▶ **Lemma 28.** *Unimportant nodes are graytone as long as the other ordinary nodes (small, sparse, important) are inactive.*

Proof. The only steps so far in which we colored vertices are Steps 1 and 4. In Step 1 we color at most $O(\log^5 \log n)$ vertices per AC and per matching M_C of each ordinary AC C. In Step 4 we only color (a subset of) the vertices in Z and one vertex per important AC (the vertex x_C for AC C). As $|Z \cap C| \leq \Delta/10$, we color at most $\Delta/10 + O(\log^5 \log n) \leq \Delta/9$ vertices in each unimportant AC.

Fix some unimportant AC C. Recall that the algorithm of [20] finds a 2.5-approximate matching, which by Lemma 11 implies that $|M_C| \geq \Delta/10$. As an unimportant AC has fewer than $\Delta/12$ nodes in $(V(M_C) \setminus C) \cap \mathcal{O}_l$, we obtain that $V(M_C) \setminus C$ contains at least $\Delta/10 - \Delta/12 = 7\Delta/60$ nodes that are not contained in $\mathcal{O}_l$. By Lemma 20, at most $O(\log^5 \log n)$ of these get colored in Step 1; denote the uncolored nodes of these by S and let $S' = N(S) \cap C$. By the earlier argument, at most $\Delta/9$ nodes of S' are already colored, that is, there exists some $v \in S'$ that is still uncolored and has an uncolored neighbor $u \notin \mathcal{O}_l$. As u is stalled to be colored later, v is gray and other nodes of the AC are grayish. ◀

▶ **Lemma 29.** *Small, sparse, and important nodes are graytone.*

Proof. By Lemma 20, each small or sparse node has slack in $G[V^* \cup \mathcal{O}]$ and is therefore gray (and stays gray until colored).

For an important AC C with triple (x_C, y_C, z_C), the node y_C is gray as x_C and z_C are same-colored. Hence, the remaining uncolored nodes of C are either already colored or graytone as they are adjacent to v. ◀

B Computing the ACD

We adapt a proof from [22] that, as stated, applies only to the case when Δ is sufficiently large. Technically, the argument differs only in that we build on [34] instead of [29] in the first step of the argument, where we compute a decomposition with weaker properties. We have opted to rephrase it, given the different constants in the definitions of these works and in order to make it more self-contained.

▶ **Lemma 5** (ACD computation [1, 22]). *For any graph $G = (V, E)$, there is a partition (*almost-clique decomposition (ACD)*) of V into sets V_{sparse} and $C_1, C_2, \ldots, C_t$ such that each node in V_{sparse} is $\Omega(\epsilon^2\Delta)$-sparse and for every $i \in [t]$,*

(i) $(1 - \varepsilon/4)\Delta \leq |C_i| \leq (1 + \varepsilon)\Delta$,

(ii) *Each $v \in C_i$ has at least $(1 - \varepsilon)\Delta$ neighbors in C_i: $|N(v) \cap C_i| \geq (1 - \varepsilon)\Delta$,*

(iii) *Each node $u \notin C_i$ has at most $(1-\varepsilon/2)\Delta$ neighbors in C_i: $|N(u) \cap C_i| \leq (1-\varepsilon/2)\Delta$.* *Further, there is an $O(1)$-round* CONGEST *algorithm to compute a valid ACD, w.h.p.*

Proof. We first use a $O(1)$-round CONGEST algorithm of [HNT22] to compute a weaker form of ACD with parameter $\varepsilon/4$.[3] Namely, it computes w.h.p. a partition $(V', D_1, D_2, \ldots, D_k)$ where nodes in V' are $\Omega(\Delta)$-sparse and we have, for each $i \in [k]$:
(a) $|D_i| \leq (1+\varepsilon/4)\Delta$, and
(b) $|N(v) \cap D_i| \geq (1-\varepsilon/4)\Delta$, for each $v \in D_i$.
What this construction does not satisfy is condition (iii).

We form a modified decomposition $(V_{sparse}, C_1, \cdots, C_k)$ as follows. For each $i \in [t]$, let C_i consist of D_i along with the nodes in V' with at least $(1-\varepsilon)\Delta$ neighbors in D_i. Let $V_{sparse} = V \setminus \cup_i C_i$. Observe that the decomposition is well-defined, as a node $u \in V'$ cannot have $(1-\varepsilon)\Delta > \Delta/2$ neighbors in more than one D_i.

We first bound from above the number of nodes added to each part C_i. Each node in D_i has at most $\varepsilon\Delta/4$ outside neighbors, so the number of edges with exactly one endpoint in D_i is at most $\varepsilon\Delta|D_i|/4 \leq \varepsilon(1+\varepsilon/4)\Delta^2/4$, using (a) to bound $|D_i|$. Each node in $C_i \setminus D_i$ is incident on at least $(1-\varepsilon)\Delta$ such edges (by definition). Thus,

$$|C_i \setminus D_i| \leq \varepsilon/4 \cdot (1+\varepsilon/4)\Delta/(1-\varepsilon) \leq \varepsilon\Delta/2 \ . \tag{2}$$

Now, (iii) holds since a node outside C_i has at most $(1-\varepsilon)\Delta$ neighbors in D_i (by the definition of C_i) and at most $|C_i \setminus D_i| \leq \epsilon\Delta/2$ other neighbors in C_i (by Equation (2)). Also, (ii) holds for nodes in D_i by (b) and for nodes in $C_i \setminus D_i$ by the definition of C_i. For the lower bound in (i), $|C_i| \geq |D_i| \geq (1-\varepsilon/4)\Delta$, by (b). For the upper bound of (i), we have $|C_i| \leq |D_i| + |C_i \setminus D_i| \leq (1+3\epsilon/4)\Delta$ (by (a) and Equation (2)).

Finally, the claim about V_{sparse} follows from the definition of V', as $V_{sparse} \subseteq V'$. ◀

[3] While such a statement is used in the paper, it is not explicitly stated. Alternatively, we may use an alternative (slower) implementation (Lemma 4.4) in [Flin et al (FGHKN22), arXiv:2301.06457]] that runs $O(\log\log n)$ rounds for $\Delta = \text{poly}(\log n)$ and still suffices for our main result. The slowdown in [FGHKN22] comes from working with sparsified graphs, while a CONGEST version also runs in $O(1)$ rounds.

Quantum Byzantine Agreement Against Full-Information Adversary

Longcheng Li
State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China

Xiaoming Sun
State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China

Jiadong Zhu[1]
State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

Abstract

We exhibit that, when given a classical Byzantine agreement protocol designed in the private-channel model, it is feasible to construct a quantum agreement protocol that can effectively handle a full-information adversary. Notably, both protocols have equivalent levels of resilience, round complexity, and communication complexity. In the classical private-channel scenario, participating players are limited to exchanging classical bits, with the adversary lacking knowledge of the exchanged messages. In contrast, in the quantum full-information setting, participating players can exchange qubits, while the adversary possesses comprehensive and accurate visibility into the system's state and messages. By showcasing the reduction from quantum to classical frameworks, this paper demonstrates the strength and flexibility of quantum protocols in addressing security challenges posed by adversaries with increased visibility. It underscores the potential of leveraging quantum principles to improve security measures without compromising on efficiency or resilience.

By applying our reduction, we demonstrate quantum advantages in the round complexity of asynchronous Byzantine agreement protocols in the full-information model. It is well known that in the full-information model, any classical protocol requires $\Omega(n)$ rounds to solve Byzantine agreement with probability one even against Fail-stop adversary when resilience $t = \Theta(n)$ [2]. We show that quantum protocols can achieve $O(1)$ rounds (i) with resilience $t < n/2$ against a Fail-stop adversary, and (ii) with resilience $t < n/(3 + \epsilon)$ against a Byzantine adversary for any constant $\epsilon > 0$, therefore surpassing the classical lower bound.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Quantum computation theory

Keywords and phrases Byzantine agreement, Quantum computation, Full-information model

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.32

Related Version *Extended Version*: `https://arxiv.org/abs/2409.01707`

Funding This work was supported in part by the National Natural Science Foundation of China Grants No. 62325210, and the Strategic Priority Research Program of Chinese Academy of Sciences Grant No. XDB28000000.

[1] Corresponding author

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 32; pp. 32:1–32:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Byzantine agreement (BA) [32], also referred to as Byzantine fault-tolerant distributed consensus, is a crucial topic in secure distributed computing. In simple terms, in a BA protocol, a group of n players who do not trust each other and possess private input bits, come to a consensus on a shared output bit, even if a subset of size t of the players are corrupted by a malicious adversary, who can force the corrupt parties to deviate from their prescribed programs during the protocol execution. The Byzantine agreement problem has been extensively researched over the past four decades, leading to numerous findings on the feasibility and potential of BA protocols in various settings [19, 11, 3].

In this paper, we focus on BA that succeeds with probability one in the full-information model, where the adversary knows the knowledge of all local variables, including quantum states if applicable. It is well known that in this model, when up to t players may be corrupted, no classical deterministic protocol can solve synchronous BA in less than $t+1$ rounds even in the presence of a Fail-stop adversary [32]. It is further proved by [5] that any classical randomized protocol requires at least expected $\tilde{\Omega}(\sqrt{n})$ rounds. Given these constraints, it is natural to ask the following question:

Can quantum communication accelerate BA in the full-information model?

The seminal work of [7] provides a confirming answer to the above question by constructing a constant round synchronous quantum BA protocol against the Byzantine adversary, surpassing the established round complexity lower bound in [5]. The protocol builds upon an expected constant round classical BA protocol introduced in [19], which is not resilient against a full-information adversary and requires a private channel. In their work, [7] proposes a quantum modification to the original classical protocol to make it robust against a full-information adversary. They achieve this by introducing a novel approach of deferring coin flips, substituting them with quantum superpositions until after the adversary has chosen his actions in a certain round. Notably, the modification does not change the structure of the original classical protocol and therefore preserves its constant round complexity. [7] also extends the synchronous quantum protocol to the asynchronous case, but with suboptimal resilience $t < n/4$.

[7] demonstrates an elegant method of reducing quantum full-information protocols to classical private-channel protocols while maintaining key attributes such as resilience, round complexity, and communication complexity. This approach offers a valuable means of evaluating the quantum advantage in the full-information model by comparing classical full-information and classical private-channel models. By highlighting the notable distinctions between these two classical models, they underscore the substantial quantum advantage inherent in the full-information domain.

In light of these findings, [7] raises the question of whether their reduction strategy could be applied to other settings, such as low round complexity asynchronous BA protocols with resilience $n/4 \leq t < n/3$, to further investigate potential quantum advantages. Unfortunately, limited progress has been made on this issue since its introduction. This paper seeks to tackle this challenge from a comprehensive viewpoint. Instead of narrowly focusing on the reduction of quantum protocols to classical protocols in a specific setting (e.g., the asynchronous protocol with resilience $n/4 \leq t < n/3$, which is better than that in [7]), our objective is to address the following question:

Is it possible to convert ***any*** *classical private-channel BA protocol to a quantum full-information BA protocol while preserving the same characteristics such as resilience, round complexity, and communication complexity?*

Table 1 Round complexity of Byzantine agreement in the full-information model.

Model	Adversary	Resilience	Classical		Quantum a)	
			Upper bound	Lower bound	Upper bound	$\mathcal{P}_C$
Sync.	Fail-stop	$t = \Theta(n)$	$\tilde{O}(\sqrt{n})$ [5]	$\tilde{\Omega}(\sqrt{n})$ [5]	$O(1)$ [7, 24]	[16]
	Byzantine	$t < n/3$	$O(n)$ [32]		$O(1)$ [7]	[19]
Async.	Fail-stop	$t = \Theta(n)$	$O(n)$ [2]	$\Omega(n)$ [2]	$O(1)$ (Our work)	[3]
	Byzantine	$t < n/4$	$\tilde{O}(n^4)$ [26]		$O(1)$ [7]	[19]
	Byzantine	$t < \frac{n}{3+\epsilon}$ b)	$\tilde{O}(n^4/\epsilon^8)$ [26]		$O(1/\epsilon)$ (Our work)	[4]
	Byzantine	$t < n/3$	$\tilde{O}(n^{12})$ [26]		$O(n)$ (Our work)	[4]

a) Every quantum protocol presented in the table is built upon some classical private-channel protocol $\mathcal{P}_C$. The last two columns of the table show the classical private-channel protocols alongside their quantum full-information equivalents for comparison and reference purposes.

b) Notice that when ϵ is a constant, the quantum upper bound is $O(1/\epsilon) = O(1)$.

1.1 Our Contribution

As our main result, we answer the above question in the affirmative by demonstrating a general reduction from a quantum full-information BA protocol to a classical private-channel BA protocol:

▶ **Theorem 1.** *Given a classical synchronous (resp. asynchronous) non-erasing BA protocol designed to counter a private-channel Fail-stop (resp. Byzantine) adversary, we can construct a quantum synchronous (resp. asynchronous) BA protocol capable of handling a full-information Fail-stop (resp. Byzantine) adversary while maintaining the same levels of resilience, round complexity, and communication complexity.*

It is crucial to emphasize that the theorem we present is applicable under the condition that the classical protocol forming the foundation of our quantum protocol is *non-erasing*, which means its security does not rely on the erasure of intermediate states. To the best of our knowledge, this criterion is met by all existing classical protocols within the scope of information-theoretic BA with probability one. For a more detailed definition of this concept, please refer to Definition 3 in Section 4 where we will provide a formal explanation. Furthermore, throughout our paper, we consistently assume that the adversary is computationally unlimited and adaptive[1], allowing it to modify its strategy based on the information acquired during the execution of the protocol.

By applying our reduction, we obtain several new quantum advantages related to round complexity in the full-information setting. As summarized in Table 1, our main result enables us to quantize existing classical private-channel protocols into some quantum full-information protocols of which the round complexity surpasses the classical lower bound in the same setting. In particular, we obtain two new quantum speedups in the asynchronous model:

- **Fail-stop model:** Section 14.3 of [3] presents a constant-round classical BA protocol with optimal resilience $t < n/2$ against the Fail-stop adversary in the private-channel setting. By applying our reduction, we obtain a constant-round quantum full-information BA protocol with $t < n/2$, while any classical full-information protocol requires $\Omega(n)$ rounds [2].

[1] Similar reductions can also be made from the quantum non-adaptive full-information model to the classical non-adaptive private-channel model.

- **Byzantine model:** For any $\epsilon > 0$, [4] presents an $O(1/\epsilon)$-round classical BA protocol with resilience $t < n/(3+\epsilon)$ against the private-channel Byzantine adversary. By applying our reduction, we obtain an $O(1/\epsilon)$-round quantum full-information BA protocol with resilience $t < n/(3+\epsilon)$. When ϵ is a constant independent of n, the quantum BA achieves constant rounds, while any classical full-information protocol requires $\Omega(n)$ rounds [2]. When $\epsilon \leq 1/n$, $\lceil n/(3+\epsilon) \rceil = \lceil n/3 \rceil$, which indicates that $t < n/(3+\epsilon)$ is equivalent to $t < n/3$. By substituting $\epsilon = 1/n$ into $O(1/\epsilon)$, we find that our quantum BA requires $O(1/\epsilon) = O(n)$ rounds. In comparison, the best known classical protocol [26] in the same setting requires $\tilde{O}(n^{12})$ rounds.

1.2 Technical Overview

We briefly explain the key ideas behind Theorem 1, especially how to quantize a classical protocol into a quantum one and how to simulate a quantum full-information adversary in the classical setting. The key idea is utilizing quantum superpositions to turn exposed randomness into hidden randomness.

A simple motivating example. Before introducing the complicated quantum full-information BA protocol against the Byzantine adversary, [7] first presents a simple quantum full-information BA protocol against the Fail-stop adversary, who can corrupt players by halting it and choosing a subset of their messages to be delivered. This simple protocol follows a common framework of reducing a BA protocol to a common-coin protocol, where all uncorrupted players need to output a common random coin with constant success probability. We will use the common-coin protocol, as demonstrated in the BA protocol against the Fail-stop adversary in [7], as a motivating example to explain the key idea of our paper. The common-coin protocol works in the quantum full-information setting and draws inspiration from a common-coin protocol in the classical private-channel setting [16]. In the following discussion, we will start by offering a brief overview of the classical private-channel protocol in [16] and explaining its limitations when confronted with a full-information adversary. We then explain how [7] effectively resolve this issue by leveraging quantum principles.

The classical private-channel protocol in [16] works as follows: (i) Each player i picks a random coin $c_i \in \{0,1\}$ and a random leader value $l_i \in [n^3]$ and then multicasts (c_i, l_i); (ii) Each player i outputs the coin c_j such that l_j is the largest leader value i receives. A private-channel Fail-stop adversary learns nothing about the values of $\{c_i\}$ and $\{l_i\}$, so the best it can do is to randomly stop t players. Since there are at least $n - t > n/2$ uncorrupted players, the largest leader falls among uncorrupted players with probability 1/2, and the probability of collision of leader values is negligible. Switching to full-information adversary, $\{c_i\}$ and $\{l_i\}$ become known to the adversary. Then the adversary can corrupt the leader and let only a subset of players receive the leader's message so that it can break the common-coin protocol. However, [7] shows that the problem can be fixed if we allow quantumness. Instead of choosing random c_i and l_i, we let player i *purify randomness*, i.e, preparing two n-qudit superposition states

$$|c_i\rangle := \frac{1}{\sqrt{2}} \left(|00\cdots 0\rangle + |11\cdots 1\rangle \right) \text{ and } |l_i\rangle := \frac{1}{\sqrt{n^3}} \sum_{l=1}^{n^3} |l, l, \ldots, l\rangle ,$$

and then distribute the n qudits of $|c_i\rangle$ and $|l_i\rangle$ among the players. In the next round, the players measure the qudits they receive and obtain the classical random coins and leader values. Although the full-information adversary can see the pure state of the system, quantum mechanics prevents it from knowing the random values before measurement. Thus this simple purified quantum protocol works against the full-information adversary.

Generalized reduction in the synchronous model. Inspired by the above example, we give a general reduction from quantum full-information BA protocols to classical private-channel BA protocols. For any classical BA protocol $\mathcal{P}_C$, the local computation of each player at round k involves (i) preparing some randomness r_k, and (ii) computing a function f to determine the decided value and messages to be sent. We construct a quantum protocol $\mathcal{P}_Q$ by modifying $\mathcal{P}_C$'s local computation to (i) preparing a quantum state $\sum_r \sqrt{\Pr[r_k = r]}\, |r\rangle$, (ii) applying a unitary U_f to compute f reversibly i.e., $U_f |v\rangle |0\rangle := |v\rangle |f(v)\rangle$ and send quantum messages.

We assume that the output of f contains a variable $d_k \in \{0, 1, \perp\}$ indicating the decided value at round k ($\perp$ if not decided yet). The player in $\mathcal{P}_Q$ will measure the corresponding quantum register of d_k and decide if $d_k \neq\perp$. In addition, to prevent a communication blowup, we also assume the output of f includes the message pattern $b_k \in \{0, 1\}^n$ where the j-th bit $b_k[j]$ indicates whether to send message to player j. $\mathcal{P}_Q$ will measure the register of b_k and send messages only to players with $b_k[j] = 1$.

Security analysis. To prove that $\mathcal{P}_Q$ is secure against a quantum full-information adversary, we follow the argument that given any quantum full-information adversary $\mathcal{A}_Q$ attacking $\mathcal{P}_Q$, we can construct a classical adversary $\mathcal{A}_C$ in the private-channel model that perfectly simulates $(\mathcal{P}_Q, \mathcal{A}_Q)$ when interacting with $\mathcal{P}_C$. However, one may question why this simulation is possible since $\mathcal{A}_Q$ is apparently more powerful than $\mathcal{A}_C$ in two aspects:

1. $\mathcal{A}_Q$ is full-information while $\mathcal{A}_C$ is private-channel.
2. $\mathcal{A}_Q$ is quantum while $\mathcal{A}_C$ is classical.

For the first problem, observe that the randomness of $\mathcal{P}_Q$ comes solely from players' measurement results of $\{b_k\}$ and $\{d_k\}$, of which the corresponding classical variables in $\mathcal{P}_C$ are also available to $\mathcal{A}_C$.[2] The pure state view of $\mathcal{P}_Q$ is fully determined by $\{b_k\}$ and $\{d_k\}$, so actually $\mathcal{A}_Q$ knows no more than $\mathcal{A}_C$ about the state of the system.

For the second problem, we first consider the Fail-stop adversary case to demonstrate why it is not a concern. The ability of a Fail-stop adversary $\mathcal{A}_Q$ is to halt players and choose a subset of their messages to be delivered, which is essentially classical. Thus $\mathcal{A}_C$ can easily simulate those actions.

The Byzantine case is trickier because a Byzantine adversary $\mathcal{A}_Q$ can apply quantum operations on the registers of corrupted players. In this case, we let $\mathcal{A}_C$ *classically simulate*[3] a quantum state on the registers of corrupted players in order to keep track of $\mathcal{A}_Q$'s actions. Moreover, when corrupted players (controlled by $\mathcal{A}_C$) send messages to uncorrupted players, they cannot simply transmit quantum messages in the manner $\mathcal{A}_Q$ does because players in $\mathcal{P}_C$ are not equipped to receive quantum information. To circumvent this challenge, we let corrupted players first measure the messages and then send the measurement outcomes, which are classical, to the uncorrupted players. Intuitively, measuring those messages will not affect the simulation because uncorrupted players always keep a copy of messages they receive. After a quantum message is sent to a uncorrupted player, corrupted players are unable to reobtain it, resulting in the message being traced out from the corrupted players' system, which is equivalent to being measured. There is still one caveat in the simulation of $\mathcal{A}_Q$ by $\mathcal{A}_C$: because $\mathcal{A}_Q$ is adaptive, it can corrupt new players during the protocol and

[2] Private-channel $\mathcal{A}_C$ knows message patterns by definition. We can also assume $\mathcal{A}_C$ knows the decided values of players because if a uncorrupted player decides in a BA protocol, all other uncorrupted players will eventually decide the same value.

[3] We assume the adversary is computationally unbounded.

reobtain the quantum messages sent to them previously, while $\mathcal{A}_C$ will only obtain collapsed classical messages when corrupting new players. To fix this, we let $\mathcal{A}_C$ maintain a copy T of the communication transcript between uncorrupted players and corrupted players. By following this approach, when $\mathcal{A}_C$ corrupts some new players, it can replicate the necessary quantum states as per the content stored in T. In this way, $\mathcal{A}_C$ can perfectly simulate $\mathcal{A}_Q$ in the classical setting.

Round and communication complexity. Our construction of $\mathcal{A}_C$ actually yields a stronger result: the probability distribution of executions in $(\mathcal{P}_Q, \mathcal{A}_Q)$ is identical to that of executions in $(\mathcal{P}_C, \mathcal{A}_C)$. This leads to the conclusion presented in Theorem 1.

Extending to the asynchronous model. Our results in the synchronous model can be extended to the asynchronous model without extra effort. The primary distinction lies in the measurement metric used; while synchronous protocols are evaluated in *rounds*, asynchronous protocols are evaluated in terms of *steps*. In one step, only one uncorrupted player receives a message, then performs local computation and possibly sends out messages. It is still feasible to purify the randomness, perform reversible computation in each step, and develop a quantum full-information protocol.

Although our results are inspired by the Fail-stop protocol in [7], our techniques are new compared with [7], especially in the Byzantine model. In the Byzantine model, [7] involves an intricate procedure of modifying the original classical protocol by replacing its classical verifiable secret sharing (VSS) component with a quantum VSS. In contrast, our approach focuses on demonstrating the efficacy of extracting purified classical randomness, a feature that is applicable to any classical protocol exhibiting a non-erasing property. Therefore, we expect our technique to have a broader range of applications.

2 Related Work

We address the construction of a quantum full-information protocol from a classical private-channel protocol. In this section, we discuss existing results in closely related contexts and provide a brief overview of their techniques.

BA protocols with private channels. The private-channel model is frequently studied in BA problems. In this model, the adversary is unable to access the contents of the messages exchanged between the participating players. A seminal work [19] presents a synchronous BA protocol that can withstand up to $t < n/3$ failures and operates within an expected constant number of rounds. Additional randomized protocols [35, 11] addressing scenarios where $n/3 \leq t < n/2$ are known, which require extra assumptions like a public-key infrastructure and a trusted dealer. Due to their dependency on these supplementary assumptions, these protocols cannot be adapted to the information-theoretic setting. In the information-theoretic setting, [3] presents an asynchronous BA protocol that can withstand up to $t < n/2$ failures while maintaining a constant running time, particularly effective against the Fail-stop adversary. For the Byzantine adversary, [1] introduces a concept called shunning verifiable secret sharing and gives an asynchronous BA protocol with optimal resilience $t < n/3$ and $O(n^2)$ running time, which is later improved to $O(n)$ by [4].

The full-information model. The full-information model, as introduced by [8], serves as a framework for investigating collective coin-flipping within a network of n players with t failures. This model has spurred a series of research efforts aimed at enhancing fault tolerance and reducing round complexity in protocols such as those proposed by [34] and [18]. [23] considers the problem of multiparty computation in the full-information model. [27] gives the first asynchronous leader election protocol in the full information model with constant success probability against a constant fraction of corrupted players. Asynchronous BA in the full-information model used to require exponential time to be solved with linear resilience [6, 10], which is recently improved to polynomial time by a sequence of works [29, 25, 26].

Quantum Byzantine protocols. Besides the work of [7], many works have applied quantum principles to Byzantine fault tolerance problems, which has led to significant advancements in the field. A key contribution is made by [20], who introduces quantum elements to Byzantine problems by addressing a weaker version called Detectable Byzantine Agreement (DBA). Their protocol involves three parties and is based on the Aharonov state. Building upon this work, [22] proposes a 3-party DBA protocol utilizing four-particle entangled qubits. Further research by [21] shows that the DBA protocol can reach any tolerance found. Other variants of the problem setting [15, 30, 33] are considered to ensure feasibility of the problem against strong Byzantine adversaries. It is also worth mentioning that a recent work [24] improves the communication complexity of the synchronous Fail-stop protocol of [7] from $O(n^2)$ to $O(n^{1+\epsilon})$ for any constant $\epsilon > 0$ while maintaining constant running time.

3 Preliminaries

3.1 Quantum Computation

In this section, we will briefly discuss quantum computation. For a more in-depth explanation, readers are encouraged to refer to [31].

In quantum computing, a *qubit* serves as the fundamental unit of quantum information, analogous to a classical bit. A *pure quantum state* in a quantum system comprising n qubits, is represented by a unit-length vector in the 2^n-dimensional Hilbert space. A commonly used basis of the space is the *computational basis* $\{|i\rangle = |i_1, i_2, \ldots, i_n\rangle : i_1, \ldots, i_n \in \{0, 1\}\}$. Then any pure state $|\psi\rangle$ can be expressed as $\sum_{i=0}^{2^n-1} \alpha_i |i\rangle$, where α_i are complex numbers known as *amplitudes*, satisfying the condition $\sum_i |\alpha_i|^2 = 1$. A *mixed quantum state*, also known as a *density matrix*, represents a probability mixture of pure states. If a quantum system is in state $|\psi_i\rangle$ with probability p_i, then its density matrix $\rho := \sum_i p_i |\psi_i\rangle \langle\psi_i|$ where $\langle\psi_i|$ denotes the conjugate transpose of $|\psi_i\rangle$. Any density matrix is Hermitian and trace one. In this paper, we also use density matrix to describe classical probability distribution: If a random variable X takes value x_i with probability p_i, then it can be described by the density matrix $\sum_i p_i |x_i\rangle \langle x_i|$.

Transformations in an n-qubit quantum system are described by unitary transformations in the 2^n-dimensional Hilbert space. Such a transformation is depicted by a unitary matrix U, which satisfies $UU^\dagger = \mathbb{I}$ where † is conjugate transpose and $\mathbb{I}$ is identity matrix. If U is applied to a pure state $|\psi\rangle$, the state becomes $U|\psi\rangle$. If U is applied to a mixed state ρ, the state becomes $U\rho U^\dagger$.

Another important operation is quantum measurement. We will only use projective measurement in our paper. A projective measurement $\mathcal{M}$ is described by a collection of orthogonal projectors $\{\Pi_i\}$ such that $\sum_i \Pi_i = \mathbb{I}$. When $\mathcal{M}$ is applied on a pure state $|\varphi\rangle$, it collapses to state $\frac{1}{\sqrt{\beta}}\Pi_i |\varphi\rangle$ with probability $\beta = \langle\varphi| \Pi_i |\varphi\rangle$. In the language of density matrix, we have $\mathcal{M}(\rho) = \sum_i \Pi_i \rho \Pi_i$. In particular, the *computational basis measurement* has projectors $\{|i\rangle \langle i| : 0 \le i < 2^n\}$. If a quantum state $\sum_i \alpha_i |i\rangle$ is measured in computational basis, it collapses to state $|i\rangle$ with probability $|\alpha_i|^2$. Measurement can also be conducted on a portion of the system or on select qubits within the system. For instance, the measurement restricted on the first qubit of a n-qubit system has projectors $\{|0\rangle\langle 0| \otimes \mathbb{I}_{2^{n-1}}, |1\rangle\langle 1| \otimes \mathbb{I}_{2^{n-1}}\}$ where $\mathbb{I}_{2^{n-1}}$ is the identity operator on the last $n-1$ qubits.

3.2 Byzantine Agreement Problem

In a Byzantine agreement problem, n distinct players labeled from 1 to n need to reach a decision on the value of a bit. Each player i inputs a bit $x_i \in \{0,1\}$ and must decide an output bit in $\{0,1\}$ that satisfies the following conditions:

1. **Agreement:** All uncorrupted players decide the same value.
2. **Validity:** If all x_i are the same bit y, then all uncorrupted players decide y.
3. **Termination:** All uncorrupted players terminate with probability 1.

The problem was introduced by Pease, Shostak and Lamport [32] in 1980. One can consider different network models, models of inter-player communication, models of local computation, and fault models. In this paper, the following models are of interest.

- **Network Models.** We will consider both synchronous network, where all messages are guaranteed to be delivered within some known time Δ from when they are sent, and asynchronous network where messages may be arbitrarily delayed.
- **Models of Inter-player Communications.** Every two players are connected by a transmit reliable[4] channel. We consider two different communication paradigms: classical and quantum. In the classical model, players can communicate classical messages, while in the quantum model, they can communicate quantum messages.[5]
- **Models of Local Computation.** In the field of Byzantine protocols, there is a common tendency to overlook the intricacies of local computations. We assume players have unbounded computational power and local memory.
- **Fault models.** We model the faulty behavior of the system by an *adversary.* The adversary can corrupt participating players and make them deviate from their prescribed programs. Once a player has been corrupted, it remains corrupted permanently. The uncorrupted players are referred to as "good" and sometimes the corrupted players are labeled as "bad". In our work, we consider the following types of adversarial behavior:
  - **Adaptive.** We will consider *adaptive* adversaries in this paper. An adaptive adversary corrupts players dynamically based on its current information at any time of the protocol.
  - **Unbound Computation.** Just like good players, the adversary has unlimited computational power and memory.
  - **Private-channel and Full-information.** We will consider both private-channel and full-information adversaries. An adversary in the private-channel model is characterized by its lack of adaptation based on the specific contents of messages exchanged within a system. Essentially, this type of adversary can only discern patterns of communication, such as the timing and players involved in message exchanges, without access to the actual message contents. By contrast, a full-information adversary possesses comprehensive knowledge of all local variables associated with the players involved in the system. In the context of the quantum model, a full information adversary knows at each point the exact pure state of the system.

[4] Messages will not be corrupted or lost during transmission.
[5] Since classical messages can also be encoded by qubits, no additional classical channels are required.

- **Fail-stop and Byzantine.** We will consider both Fail-stop and Byzantine adversaries. The players corrupted by the Fail-stop adversary will no longer take part in the protocol. We remark that a private-channel Fail-stop adversary *cannot* read the local memory of corrupted players.[6] However, the players corrupted by a Byzantine adversary can deviate arbitrarily from the protocol.

We are interested in several metrics that measure the performance of BA protocols:

- **Resilience:** the maximum number of parties that can be corrupted within the protocol.
- **Round Complexity:** Assume there is a virtual "global clock" within the network that is not accessible to any player. In this context, the term *delay* refers to the time taken from sending a message to its reception. The *number of rounds*[7] in an execution refers to the total execution time divided by the longest message delay. The *round complexity* of a protocol $\mathcal{P}$ is defined as the maximum expected number of rounds in $\mathcal{P}$'s executions, considering all inputs and potential adversaries.
- **Communication Complexity:** the maximum expected number of messages sent by good players throughout the protocol, considering all inputs and potential adversaries.

3.3 Helper lemmas

The following two lemmas will be used, of which the proofs are given in Appendix A.

▶ **Lemma 1.** *Let $\mathcal{M}$ be the computational basis measurement of a Hilbert space $\mathcal{H}$. Then $\mathcal{M}$ commutes with*

1. *any permutation unitary U acting on $\mathcal{H}$;*
2. *any orthogonal projector Π on $\mathcal{H}$ in computational basis.*

▶ **Lemma 2.** *Let G be good players' registers, B be bad players' registers. Initially G and B are independent and then they make quantum communication for several rounds. Assume G keeps a local copy of the communication transcript between G and B. Then the pure state of the system GB can be written as $\sum_m \alpha_m |m, \phi_m\rangle_{\mathsf{G}} \otimes |\psi_m\rangle_{\mathsf{B}}$ where $|m\rangle$ are the communication transcripts, $|\phi_m\rangle$ are states of G besides the communication transcripts, and $|\psi_m\rangle$ are states of B.*

4 Proof of Main Theorem

In this section, we prove our main theorem by giving a general reduction from quantum full-information BA protocols to classical private-channel protocols.

▶ **Theorem 1.** *Given a classical synchronous (resp. asynchronous) non-erasing BA protocol designed to counter a private-channel Fail-stop (resp. Byzantine) adversary, we can construct a quantum synchronous (resp. asynchronous) BA protocol capable of handling a full-information Fail-stop (resp. Byzantine) adversary while maintaining the same levels of resilience, round complexity, and communication complexity.*

Our reduction requires a "non-erasing" property of classical private-channel protocols:

[6] Some BA protocols consider a stronger Fail-stop adversary who can read the memory of corrupted players, but our Theorem 1 still applies to those protocols because we only require security against a weaker Fail-stop adversary.

[7] In the synchronous model, this definition is equivalent to the number of synchronous rounds during the execution.

▶ **Definition 3** (Non-erasing BA protocol). *In the context of a classical BA protocol denoted as $\mathcal{P}$, each computational step performed by a player can be seen as the evaluation of a function $f(s)$ where s is the internal state of the player. Consider a modified protocol, denoted as $\mathcal{P}'$, which follows the structure of $\mathcal{P}$ except that players in $\mathcal{P}'$ keep a copy of their previous state s in their local memory subsequent to each evaluation of $f(s)$.*

A BA protocol such as $\mathcal{P}$ is called non-erasing *if the adjusted protocol $\mathcal{P}'$ maintains the characteristics of being a BA protocol while preserving the same level of resilience, round and communication complexity as $\mathcal{P}$.*

To the best of our knowledge, this non-erasing property is considered a reasonable assumption as it is met by all existing protocols within the scope of information-theoretic BA with probability one, e.g., [16, 19, 3, 4]. Beyond our scope, there exist BA protocols requiring the ability to securely erase intermediate secrets, often referred to as the *memory-erasure model* [17]. Those protocols either rely on cryptographic assumptions [13] or succeed only with high probability [28].

The rest of this section is to prove Theorem 1. For simplicity, we will only give a full proof for the synchronous model (Section 4.1) and then briefly discuss how to extend it to the asynchronous case (Section 4.2).

4.1 Synchronous Model

In this subsection, we prove Theorem 1 for the synchronous model. Without loss of generality, we assume a synchronous classical non-erasing private-channel BA protocol $\mathcal{P}_C$ has the following normal form.

Classical protocol $\mathcal{P}_C$. Let k denote the round number, $m_k^{(i,j)}$ denote the message sent from i to j and $m_k'^{(i,j)}$ denote a copy of $m_k^{(i,j)}$ to be kept by i, $b_k^{(i,j)} \in \{0,1\}$ denote the message pattern which is 1 if $m_k^{(i,j)}$ is non-empty, and $d^{(i)} \in \{0,1,\perp\}$ denote the decided value of i ($\perp$ if not decided yet). We also use $m_k^{(*,i)}$ to denote the vector $\left(m_k^{(1,i)}, m_k^{(2,i)}, \ldots, m_k^{(n,i)}\right)$ and $m_k'^{(i,*)}, m_k^{(i,*)}, b_k^{(i,*)}$ are defined similarly. At round k, player i on input x_i executes the following steps.

$\mathcal{P}_C$ for player i at round k

1. Receive messages $m_{k-1}^{(*,i)}$ from other players if $k > 1$.
2. Sample randomness $r_k^{(i)}$.
3. Compute a function $f_P : \mathrm{View}_k^{(i)} \to \left(m_k^{(i,*)}, m_k'^{(i,*)}, b_k^{(i,*)}, d_k^{(i)}\right)$ where[a]

$$\mathrm{View}_k^{(i)} := \begin{cases} \left(i, x_i, r_1^{(i)}\right) & \text{if } k = 1 \\ \left(\mathrm{View}_{k-1}^{(i)}, m_{k-1}'^{(i,*)}, m_{k-1}^{(*,i)}, r_k^{(i)}\right) & \text{otherwise} \end{cases}.$$

4. If the decided value $d_k^{(i)} \neq \perp$, output value $d_k^{(i)}$ and terminate.[b]
5. For $j \in [n]$, send messages $m_k^{(i,j)}$ to player j if $b_k^{(i,j)} = 1$.

[a] Keeping $\mathrm{View}_k^{(i)}$ in memory does not lose generality beacuse $\mathcal{P}_C$ is non-erasing.

[b] We assume a player decides and terminates at the same time, since otherwise we can always defer the decision until the player terminates.

Then we construct a quantum BA protocol $\mathcal{P}_Q$ by quantizing $\mathcal{P}_C$ as follows. The essential idea is to purify the local randomness, compute everything reversibly, and do as little measurement as possible. In this way, only a superposition of all possible local information is revealed to the quantum full-information adversary. Formally,

Quantum protocol $\mathcal{P}_Q$. Let k denote the round number, $\mathsf{M}_k^{(i,j)}, \mathsf{M}_k'^{(i,j)}, \mathsf{B}_k^{(i,j)}, \mathsf{D}_k^{(i)}, \mathsf{R}_k^{(i)}$ denote the quantum registers holding the message from player i to player j, the copy of the message, the message pattern, the decided value of player i, and the randomness of player i respectively. At round k, player i on input x_i executes the following steps.

$\mathcal{P}_Q$ for player i at round k

1. Receive quantum messages $\mathsf{M}_{k-1}^{(*,i)}$ from other players if $k > 1$.
2. Prepare a quantum state $\sum_r \sqrt{\Pr[r_k^{(i)} = r]}\,|r\rangle$ in a new quantum register $\mathsf{R}_k^{(i)}$.
3. Let $U_P^{(i)}$ denote the unitary $|v\rangle\,|y\rangle \to |v\rangle\,|y + f_P(v)\rangle$ which reversibly computes function f_P. Execute U_P on register $\mathsf{View}_k^{(i)}$ and an empty ancilla register $\mathsf{A}_k^{(i)} := \left(\mathsf{M}_k^{(i,*)}, \mathsf{M}_k'^{(i,*)}, \mathsf{B}_k^{(i,*)}, \mathsf{D}_k^{(i)}\right)$ where
$$\mathsf{View}_k^{(i)} := \begin{cases} |i\rangle\langle i| \otimes |x_i\rangle\langle x_i| \otimes \mathsf{R}_1^{(i)} & \text{if } k = 1 \\ \left(\mathsf{View}_{k-1}^{(i)}, \mathsf{M}_{k-1}'^{(i,*)}, \mathsf{M}_{k-1}^{(*,i)}, \mathsf{R}_k^{(i)}\right) & \text{otherwise} \end{cases}.$$
4. Measure register $\mathsf{D}_k^{(i)}$. If the result $d_k^{(i)} \neq \perp$, output $d_k^{(i)}$ and terminate.
5. For each $j \in [n]$, measure $\mathsf{B}_k^{(i,j)}$. If the result $b_k^{(i,j)} = 1$, send the $\mathsf{M}_k^{(i,j)}$ to player j.

In the rest of this subsection, for both Fail-stop and Byzantine cases, we prove that $\mathcal{P}_Q$ is a quantum full-information BA protocol with the same resilience, round and communication complexity as $\mathcal{P}_C$. The proof follows the argument that assuming there is quantum full-information adversary $\mathcal{A}_Q$ attacking $\mathcal{P}_Q$, we can construct a classical adversary $\mathcal{A}_C$ in the private-channel model attacking $\mathcal{P}_C$.

4.1.1 Fail-stop adversary

Without loss of generality, we assume the adversary launches attacks at the beginning of each round for both $\mathcal{P}_C$ and $\mathcal{P}_Q$. The Fail-stop adversary has the ability to adaptively halt some players and choose only a subset of their messages in this round to be received. Now consider a quantum full-information Fail-stop adversary $\mathcal{A}_Q$ attacking $\mathcal{P}_Q$, which can be formalized as follows.

Quantum full-information adversary $\mathcal{A}_Q$. Assume $\mathcal{A}_Q$ samples its randomness r_A before the protocol starts. Then at round k, $\mathcal{A}_Q$ first chooses the set of corrupted players S_k up to round k such that $|S_k| \le t$ and $S_k \supseteq S_{k-1}$, and then $\mathcal{A}_Q$ decides only a subset of $S_k \setminus S_{k-1}$'s messages to be sent. Here, we model the message exchanging step as a permutation unitary V_k which swaps the registers $\mathsf{M}_k^{(i,j)}$ and the receiving register of player j for $i, j \in [n]$. Then $\mathcal{A}_Q$'s attack can be modeled by choosing an appropriate V_k. Thus $\mathcal{A}_Q$ can be viewed as a function $f_A : r_A, \mathsf{View}_1, \mathsf{View}_2, \ldots, \mathsf{View}_{k-1} \to (S_k, V_k)$ where View_j is the pure state view of the system at round j.

Let $b_j := (b_j^{(1,1)}, \ldots, b_j^{(n,n)})$ and $d_j := (d_j^{(1)}, \ldots, d_j^{(n)})$. Observe that the randomness of the system comes only from classical variables r_A, $\{b_j\}$, $\{d_j\}$, so the pure state View_k is fully determined by those variables. Thus there exists a function f_V such that

$f_V(r_A, b_1, d_1, b_2, d_2, \ldots, b_k, d_k) = \mathsf{View}_k$. Since the variables $\{b_j\}$ and $\{d_j\}$ in $\mathcal{P}_C$ are also available to classical private-channel adversaries, now we construct a classical private-channel adversary $\mathcal{A}_C$ attacking $\mathcal{P}_C$.

Classical adversary $\mathcal{A}_C$ in the private-channel model. First sample the same randomness r_A as $\mathcal{A}_Q$ before the protocol starts. Then at round k, compute its action by the following steps.

1. For each $j \in [k-1]$, compute quantum state $|\psi_j\rangle := f_V(r_A, b_1, d_1, b_2, d_2, \ldots, b_j, d_j)$.
2. Compute action $(S_k, V_k) := f_A(r_A, |\psi_1\rangle, |\psi_2\rangle, \ldots, |\psi_{k-1}\rangle)$.

Then we prove that $\mathcal{A}_C$ perfectly simulates the execution of $(\mathcal{P}_Q, \mathcal{A}_Q)$ when interacting with $\mathcal{P}_C$, which is characterized by Lemma 5.

▶ **Definition 4.** *A k-round execution $\mathcal{E}$ of $(\mathcal{P}_C, \mathcal{A}_C)$ is a sequence $r_A, (b_1, d_1), (b_2, d_2), \ldots, (b_k, d_k)$. $\mathcal{E}$ is also a k-round execution of $(\mathcal{P}_Q, \mathcal{A}_Q)$ since the pure states of the system at each round can completely determined by $\mathcal{E}$ using f_V.*

▶ **Lemma 5.** *Any k-round execution $\mathcal{E}$ occurs in $(\mathcal{P}_Q, \mathcal{A}_Q)$ and $(\mathcal{P}_C, \mathcal{A}_C)$ with the same probability. Furthermore, if the pure state after $\mathcal{E}$ in $(\mathcal{P}_Q, \mathcal{A}_Q)$ is $\sum_u \alpha_u |u\rangle$, then the distribution of the system's possible states after $\mathcal{E}$ in $(\mathcal{P}_C, \mathcal{A}_C)$ conditioned on $\mathcal{E}$ is $\sum_u |\alpha_u|^2 |u\rangle\langle u|$.*[8]

Proof. See Appendix B. ◀

By the above lemma, we have:

▶ **Proposition 6.** *In the synchronous Fail-stop model, given a non-erasing classical private-channel BA protocol $\mathcal{P}_C$, there exists a quantum full-information BA protocol $\mathcal{P}_Q$ with the same resilience, round and communication complexity as $\mathcal{P}_C$.*

Proof. Assuming there exists an adversary $\mathcal{A}_Q$ that can cause an inconsistent, invalid or non-terminating execution $\mathcal{E}$ in $(\mathcal{P}_Q, \mathcal{A}_Q)$ with probability $p > 0$ by corrupting $\le t$ players, then $\mathcal{E}$ also occurs in $(\mathcal{P}_C, \mathcal{A}_C)$ with probability p by Lemma 5, which gives a contradiction. Thus the resilience of $\mathcal{P}_Q$ is at least the resilience of $\mathcal{P}_C$.

Given an execution $\mathcal{E}$, let $|\mathcal{E}|$ be the number of rounds of $\mathcal{E}$, and $\mathrm{CC}(\mathcal{E}) := \sum_{k=1}^{|\mathcal{E}|} \sum_{i \in \bar{S}_k, j \in [n]} b_k^{(i,j)}$ denote the number of messages in $\mathcal{E}$. Then by Lemma 5,[9]

$$\begin{aligned}
\mathrm{RC}(\mathcal{P}_Q) &:= \max_{\mathcal{A}_Q} \mathop{\mathbb{E}}_{\text{execution } \mathcal{E}} \Pr[\mathcal{E} \in (\mathcal{P}_Q, \mathcal{A}_Q)] \cdot |\mathcal{E}| \\
&= \max_{\mathcal{A}_C \in \mathcal{Q}} \mathop{\mathbb{E}}_{\text{execution } \mathcal{E}} \Pr[\mathcal{E} \in (\mathcal{P}_C, \mathcal{A}_C)] \cdot |\mathcal{E}| \le \mathrm{RC}(\mathcal{P}_C), \\
\mathrm{CC}(\mathcal{P}_Q) &:= \max_{\mathcal{A}_Q} \mathop{\mathbb{E}}_{\text{execution } \mathcal{E}} \Pr[\mathcal{E} \in (\mathcal{P}_Q, \mathcal{A}_Q)] \cdot \mathrm{CC}(\mathcal{E}) \\
&= \max_{\mathcal{A}_C \in \mathcal{Q}} \mathop{\mathbb{E}}_{\text{execution } \mathcal{E}} \Pr[\mathcal{E} \in (\mathcal{P}_C, \mathcal{A}_C)] \cdot \mathrm{CC}(\mathcal{E}) \le \mathrm{CC}(\mathcal{P}_C)
\end{aligned}$$

where $\mathrm{RC}(\cdot)$ denotes round complexity, $\mathrm{CC}(\cdot)$ denotes communication complexity, and $\mathcal{Q}$ denotes the set of classical private-channel adversaries that are constructed from some quantum full-information adversary in the beyond way. ◀

[8] We use density matrix to represent classical probability distribution. See Section 3.1 for details.

[9] For simplicity, we can assume players' input of is chosen by the adversary, so there is no need to take maximum over the input.

4.1.2 Byzantine adversary

For the Byzantine case, we also assume the adversary launches attacks at the beginning of each round. Unlike the Fail-stop adversary, the Byzantine adversary can manipulate corrupted players in an arbitrary way. Now consider a quantum full-information Byzantine adversary $\mathcal{A}_Q$ attacking $\mathcal{P}_Q$, which can be formalized as follows.

Quantum full-information adversary $\mathcal{A}_Q$. Assume $\mathcal{A}_Q$ samples its randomness r_A before the protocol starts. Let S_k denote the corrupted players up to round k such that $|S_k| \leq t, S_k \supseteq S_{k-1}$, and $\bar{S}_k := [n] \setminus S_k$ denote good players. Here, we model the message-exchanging step differently from the Fail-stop case. When player j receives the message from i, the register $\mathsf{M}_k^{(i,j)}$ is simply appended to j's workspace. Then at round k, $\mathcal{A}_Q$ acts as follows.

1. First let current corrupted players S_{k-1} receive all the messages sent to them.
2. Apply arbitrary quantum operation on S_{k-1}, which can be decomposed as a unitary U_k and a measurement operator $\mathcal{M}_k$ on the registers of S_{k-1} by Stinespring dilation theorem.[10] Let a_k denote the measurement outcome.
3. Choose an enlarged set S_k of corrupted players and corrupt $S_k \setminus S_{k-1}$.
4. Apply arbitrary quantum operation on S_k, which can be decomposed as applying a unitary U'_k and a measurement operator $\mathcal{M}'_k$ on the registers of S_k. Let a'_k denote the measurement outcome.

We remark that step 4 is necessary because an adaptive adversary can decide to corrupt a player i and stop (or change) the message just sent by i in step 5 of the previous round.

Similar to the Fail-stop case, the adversary's operations $U_k, \mathcal{M}_k, U'_k, \mathcal{M}'_k$ and the corrupted set S_k are all functions of randomness r_A and the system's pure states at each step. And the system's pure states can be fully determined by classical variables $r_A, \{a_j\}, \{a'_j\}, \{b_j\}$ and $\{d_j\}$. Thus we can define two functions g_A and f_A such that

$$g_A\left(r_A, a_1, a'_1, b_1, d_1, \ldots, a_{k-1}, a'_{k-1}, b_{k-1}, d_{k-1}\right) = (U_k, \mathcal{M}_k), \text{ and}$$
$$f_A\left(r_A, a_1, a'_1, b_1, d_1, \ldots, a_{k-1}, a'_{k-1}, b_{k-1}, d_{k-1}, a_k\right) = (S_k, U'_k, \mathcal{M}'_k).$$

Additionally, we define $\Phi\left(r_A, a_1, a'_1, b_1, d_1, \ldots, a_{k-1}, a'_{k-1}, b_{k-1}, d_{k-1}, a_k\right)$ to be the system's pure state right after step 3 of $\mathcal{A}_Q$ at round k. Then by Lemma 2, we have

$$\Phi\left(r_A, a_1, a'_1, b_1, d_1, \ldots, a_{k-1}, a'_{k-1}, b_{k-1}, d_{k-1}, a_k\right) = \sum_m \alpha_m \left|m, \phi_m\right\rangle_{\bar{S}_k} \left|\psi_m\right\rangle_{S_k} \tag{1}$$

where $|m\rangle$ are the copy of messages between $\bar{S}_k$ and S_k kept by $\bar{S}_k$, $|\phi_m\rangle$ are states of $\bar{S}_k$ besides the copy, and $|\psi_m\rangle$ are states of S_k.

Since classical variables $\{b_k\}, \{d_k\}$ in $\mathcal{P}_C$ are also available to the adversary in the private-channel model, we can construct a classical Byzantine adversary $\mathcal{A}_C$ attacking $\mathcal{P}_C$ as follows.

[10] Stinespring dilation theorem [14] states that for any quantum operation $\mathcal{E}$, there exists a unitary U and an environment space E such that $\mathcal{E}(\rho) = \mathrm{Tr}_{\mathsf{E}}\left(U(\rho \otimes |0\rangle \langle 0|_{\mathsf{E}})U^\dagger\right)$. The partial trace Tr_{E} is equivalent to measuring E. We can assume players start with large enough empty workspace so there is no need to append new ancilla space in order to perform U.

Classical adversary $\mathcal{A}_C$ in the private-channel model. First sample the same randomness r_A as $\mathcal{A}_Q$ before the protocol starts. During the protocol, $\mathcal{A}_C$ maintains a communication transcript T between good players $\bar{S}_k$ and bad players S_k. Also, $\mathcal{A}_C$ *classically simulates* a quantum state of the registers of S_k, which is denoted by $|\varphi_k\rangle$ after round k. At round k, $\mathcal{A}_C$ acts as follows.

1. Let S_{k-1} receive all the messages $m_{k-1}^{(*,S_{k-1})}$ sent to them and record in T.
2. Compute $(U_k, \mathcal{M}_k)$ by g_A. Then apply U_k and $\mathcal{M}_k$ on $|\varphi_{k-1}\rangle \otimes |m_{k-1}^{(*,S_{k-1})}\rangle$ and obtain the measurement outcome a_k.
3. Compute $(S_k, U_k', \mathcal{M}_k')$ by f_A. Corrupt players S_k and update T as the communication transcript between new sets $\bar{S}_k$ and S_k. Then according to T, $\mathcal{A}_C$ discards old state $|\varphi_{k-1}\rangle$ and simulates a new state $|\psi_T\rangle$ which is defined in Eq. (1).
4. Apply U_k' and $\mathcal{M}_k'$ to $|\psi_T\rangle$ and obtain measurement outcome a_k'. Then apply a computational basis measurement $\mathcal{M}_{msg}$ on messages to be sent from S_k to $\bar{S}_k$ and add those messages to T. Let $|\varphi_k\rangle$ be the pure state after applying U_k', $\mathcal{M}_k'$ and $\mathcal{M}_{msg}$.

▶ **Definition 7.** *A k-round execution $\mathcal{E}$ of $(\mathcal{P}_C, \mathcal{A}_C)$ is a sequence $r_A, (a_1, a_1', b_1, d_1), \ldots, (a_k, a_k', b_k, d_k)$. $\mathcal{E}$ is also a k-round execution of $(\mathcal{P}_Q, \mathcal{A}_Q)$ since the pure states of the system can be determined by $\mathcal{E}$.*

▶ **Lemma 8.** *Any k-round execution $\mathcal{E}$ occurs in $(\mathcal{P}_Q, \mathcal{A}_Q)$ and $(\mathcal{P}_C, \mathcal{A}_C)$ with the same probability. Furthermore, if the pure state in $(\mathcal{P}_Q, \mathcal{A}_Q)$ after $\mathcal{E}$ is $|Q_k\rangle$, then the distribution of system's state in $(\mathcal{P}_C, \mathcal{A}_C)$ after $\mathcal{E}$ is $C_k := \mathcal{M}_{\bar{S}_k}(|Q_k\rangle\langle Q_k|)$ where $\mathcal{M}_{\bar{S}_k}$ is the computational basis measurement on good players $\bar{S}_k$'s registers.*[11]

Proof. See Appendix C. ◀

By the above lemma, we conclude Theorem 1 for the synchronous Byzantine case, which can be proven the same way as the Fail-stop case (Proposition 6).

▶ **Proposition 9.** *In the synchronous Byzantine model, given a non-erasing classical private-channel BA protocol $\mathcal{P}_C$, there exists a quantum full-information BA protocol $\mathcal{P}_Q$ with the same resilience, round and communication complexity as $\mathcal{P}_C$.*

4.2 Asynchronous Model

The techniques used in the proof above can be extended to the asynchronous model as well. However, a key distinction lies in the terminology used to characterize the execution: while the synchronous model employs “rounds”, the asynchronous model employs “steps”. In this context, a step involves a single good player receiving only one message, carrying out computations, and potentially transmitting messages. The order in which players receive messages is determined by the adversary. For simplicity, we assume that each player initially receives its input as its first message, and each message contains the sender's ID.

In alignment with the synchronous model, our approach involves first giving a normal form to any asynchronous classical non-erasing private-channel BA protocol $\mathcal{P}_C$ and then quantizing it into a quantum protocol $\mathcal{P}_Q$ against the full-information adversary.

[11] Density matrix C_k represents a distribution of system's states with classical $\bar{S}_k$ and quantum S_k. It is classically feasible because S_k's quantum state is classically simulated, and the correlation between $\bar{S}_k$ and S_k is classical, i.e., there is no quantum entanglement.

Classical protocol $\mathcal{P}_C$. Each player is activated each time it receives a message. Let $\pi_k^{(i)}$ denote the k-th message player i receives, where the first message $\pi_1^{(i)}$ is its input x_i. The notations $m_k^{(i,j)}, m_k'^{(i,j)}, b_k^{(i)}, d_k^{(i)}$ are defined similarly as in synchronous model (Section 4.1).

$\mathcal{P}_C$ for player i upon receiving the k-th message $\pi_k^{(i)}$

1. Sample randomness $r_k^{(i)}$.
2. Compute a function $f_P : \mathrm{View}_k^{(i)} \to \left(m_k^{(i,*)}, m_k'^{(i,*)}, b_k^{(i,*)}, d_k^{(i)}\right)$ where
$$\mathrm{View}_k^{(i)} := \begin{cases} \left(i, x_i, r_1^{(i)}\right) & \text{if } k = 1 \\ \left(\mathrm{View}_{k-1}^{(i)}, m_{k-1}'^{(i,*)}, \pi_k^{(i)}, r_k^{(i)}\right) & \text{otherwise} \end{cases}.$$
3. If the decided value $d_k^{(i)} \neq \perp$, output value $d_k^{(i)}$ and terminate.
4. For $j \in [n]$, send messages $m_k^{(i,j)}$ to player j if $b_k^{(i,j)} = 1$.

Quantum protocol $\mathcal{P}_Q$. Each player is activated each time it receives a message. Let $\mathsf{\Pi}_k^{(i)}$ denote the k-th quantum message player i receives. The notations $\mathsf{M}_k^{(i,j)}, \mathsf{M}_k'^{(i,j)}, \mathsf{B}_k^{(i,j)}, \mathsf{D}_k^{(i)}$ are defined similarly as in synchronous model (Section 4.1). We remark that $\mathsf{\Pi}_k^{(i)}$ is an alias of register $\mathsf{M}_{k'}^{(j',i)}$ for some j', k'.

$\mathcal{P}_Q$ for player i upon receiving the k-th message $\mathsf{\Pi}_k^{(i)}$

1. Prepare a quantum state $\sum_r \sqrt{\Pr[r_k^{(i)} = r]}\,|r\rangle$ in a new quantum register $\mathsf{R}_k^{(i)}$.
2. Let $U_P^{(i)}$ denote the unitary $|v\rangle\,|y\rangle \to |v\rangle\,|y + f_P(v)\rangle$ which reversibly computes function f_P. Execute $U_P^{(i)}$ on register $\mathsf{View}_k^{(i)}$ and an empty ancilla register $\mathsf{A}_k^{(i)} := \left(\mathsf{M}_k^{(i,*)}, \mathsf{M}_k'^{(i,*)}, \mathsf{B}_k^{(i,*)}, \mathsf{D}_k^{(i)}\right)$ where
$$\mathsf{View}_k^{(i)} := \begin{cases} |i\rangle\langle i| \otimes |x_i\rangle\langle x_i| \otimes \mathsf{R}_1^{(i)} & \text{if } k = 1 \\ \left(\mathsf{View}_{k-1}^{(i)}, \mathsf{M}_{k-1}'^{(i,*)}, \mathsf{\Pi}_k^{(i)}, \mathsf{R}_k^{(i)}\right) & \text{otherwise} \end{cases}.$$
3. Measure register $\mathsf{D}_k^{(i)}$. If the result $d_k^{(i)} \neq \perp$, output $d_k^{(i)}$ and terminate.
4. For each $j \in [n]$, measure $\mathsf{B}_k^{(i,j)}$. If the result $b_k^{(i,j)} = 1$, send the $\mathsf{M}_k^{(i,j)}$ to player j.

Then we claim that $\mathcal{P}_Q$ is a quantum BA protocol against the quantum full-information adversary in the asynchronous model with the same round and communication complexity as $\mathcal{P}_C$. The proof is almost the same as the synchronous case, so we only sketch the proof here.

Assuming there is a quantum full-information Fail-stop (resp. Byzantine) adversary $\mathcal{A}_Q$ attacking $\mathcal{P}_Q$, we can construct a classical private-channel Fail-stop (resp. Byzantine) adversary $\mathcal{A}_C$ attacking $\mathcal{P}_C$ as in Section 4.1.1 (resp. Section 4.1.2). Then we can define execution execution in the asynchronous model.

▶ **Definition 10** (Informal). *A k-step execution $\mathcal{E}$ is defined to be a sequence r_A, (a_1, b_1, d_1), $(a_2, b_2, d_2), \ldots, (a_k, b_k, d_k)$ where r_A is the adversary's randomness, a_j is some classical information the adversary obtains at step j, $b_j \in \{0,1\}^n$ is the message pattern, and $d_j \in \{0, 1, \perp\}$ is the decided value of the player activated at step j.*

Then similar to Lemma 5 (resp. Lemma 8), we prove that any execution occurs in $(\mathcal{P}_Q, \mathcal{A}_Q)$ with the same probability.

▶ **Lemma 11** (Informal). *Any k-step execution $\mathcal{E}$ occurs in $(\mathcal{P}_Q, \mathcal{A}_Q)$ and $(\mathcal{P}_C, \mathcal{A}_C)$ with the same probability.*

Since the information contained in an execution $\mathcal{E}$ fully determines the properties, number of rounds, and number of messages of the protocol, we can conclude Theorem 1 in the asynchronous case. This can be proven similarly to the synchronous case (Proposition 6 and Proposition 9).

▶ **Proposition 12.** *In the asynchronous Fail-stop (or Byzantine) model, given a non-erasing classical private-channel BA protocol $\mathcal{P}_C$, there exists a quantum full-information BA protocol $\mathcal{P}_Q$ with the same resilience, round and communication complexity as $\mathcal{P}_C$.*

Proof of Theorem 1. Theorem 1 can be obtained by integrating the results from Proposition 6, Proposition 9 and Proposition 12. ◀

5 Discussions

In this paper, we present a general reduction from quantum full-information BA protocols to classical private-channel BA protocols that preserves resilience, round and communication complexity. Utilizing this reduction, we make progress towards the open question posed by [7] of whether quantum BA can achieve $O(1)$ round complexity and optimal resilience $t < n/3$ simultaneously in the asynchronous full-information model. We show that $O(1)$ round complexity and suboptimal resilience $t < n/(3 + \epsilon)$ is possible for any constant $\epsilon > 0$. Our reduction also suggests that designing a better classical private-channel protocol may finally lead to the resolution of this open question.

There are several interesting directions for future research. Firstly, it would be valuable to explore whether the reverse of our reduction is possible, i.e., whether any quantum full-information BA protocol can be converted to a classical private-channel BA protocol without compromising key attributes like resilience. Existing techniques in this paper do not apply due to the ability of good players to employ quantum operations. Secondly, it is worth considering the potential generalization of our results to less strict models, such as BA that terminates only with high probability [12, 28], or BA that requires erasing intermediate states [28, 13]. Thirdly, it is worthwhile to explore the potential for developing BA protocols with improved performance by granting quantum players the ability to utilize private memory, thereby shifting the adversary from a position of full-information to one of limited knowledge. This model presents an intriguing opportunity for innovation, especially considering the existence of quantum key distribution in such a framework [9]. Finally, while our primary focus is on addressing the BA problem as it stands as a fundamental challenge in this field, we anticipate that our methods can also be applied to other fault-tolerant distributed computing tasks like coin toss and leader election.

References

1 Ittai Abraham, Danny Dolev, and Joseph Y Halpern. An almost-surely terminating polynomial protocol for asynchronous byzantine agreement with optimal resilience. In *Proceedings of the Twenty-seventh ACM symposium on Principles of Distributed Computing*, pages 405–414, 2008. doi:10.1145/1400751.1400804.

2 Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *Journal of the ACM (JACM)*, 55(5):1–26, 2008. doi:10.1145/1411509.1411510.

3 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley & Sons, Inc., Hoboken, NJ, USA, 2004.

4 Laasya Bangalore, Ashish Choudhury, and Arpita Patra. Almost-surely terminating asynchronous byzantine agreement revisited. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 295–304, 2018. URL: `https://dl.acm.org/citation.cfm?id=3212735`.

5 Ziv Bar-Joseph and Michael Ben-Or. A tight lower bound for randomized synchronous consensus. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 193–199, New York, NY, USA, 1998. Association for Computing Machinery. `doi:10.1145/277697.277733`.

6 Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pages 27–30. ACM, 1983. `doi:10.1145/800221.806707`.

7 Michael Ben-Or and Avinatan Hassidim. Fast quantum byzantine agreement. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 481–485, New York, NY, USA, 2005. Association for Computing Machinery. `doi:10.1145/1060590.1060662`.

8 Michael Ben-Or and Nathan Linial. Collective coin flipping, robust voting schemes and minima of banzhaf values. In *26th Annual Symposium on Foundations of Computer Science (SFCS 1985)*, pages 408–416, 1985. `doi:10.1109/SFCS.1985.15`.

9 Charles H Bennett and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science*, 560:7–11, 2014. `doi:10.1016/j.tcs.2014.05.025`.

10 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. `doi:10.1016/0890-5401(87)90054-X`.

11 Gabriel Bracha. An $O(\log n)$ expected rounds randomized byzantine generals protocol. *J. ACM*, 34(4):910–920, October 1987. `doi:10.1145/31846.42229`.

12 Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, pages 42–51, 1993. `doi:10.1145/167088.167105`.

13 Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019. `doi:10.1016/j.tcs.2019.02.001`.

14 Man-Duen Choi. Completely positive linear maps on complex matrices. *Linear Algebra and its Applications*, 10(3):285–290, 1975.

15 Vicent Cholvi. Quantum byzantine agreement for any number of dishonest parties. *Quantum Information Processing*, 21(4):151, 2022. `doi:10.1007/s11128-022-03492-y`.

16 Benny Chor, Michael Merritt, and David B Shmoys. Simple constant-time consensus protocols in realistic failure models. *Journal of the ACM (JACM)*, 36(3):591–614, 1989. `doi:10.1145/65950.65956`.

17 Thaddeus Dryja, Quanquan C Liu, and Neha Narula. A lower bound for byzantine agreement and consensus for adaptive adversaries using vdfs. *arXiv preprint arXiv:2004.01939*, abs/2004.01939, 2020. `arXiv:2004.01939`, `doi:10.48550/arXiv.2004.01939`.

18 U. Feige. Noncryptographic selection protocols. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pages 142–152, 1999. `doi:10.1109/SFFCS.1999.814586`.

19 Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, 1997. `doi:10.1137/S0097539790187084`.

20 Matthias Fitzi, Nicolas Gisin, and Ueli Maurer. Quantum solution to the byzantine agreement problem. *Phys. Rev. Lett.*, 87:217901, November 2001. `doi:10.1103/PhysRevLett.87.217901`.

21 Matthias Fitzi, Daniel Gottesman, Martin Hirt, Thomas Holenstein, and Adam Smith. Detectable byzantine agreement secure against faulty majorities. In *Proceedings of the Twenty-First*

Annual Symposium on Principles of Distributed Computing, PODC '02, pages 118–126, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/571825.571841.

22 Sascha Gaertner, Mohamed Bourennane, Christian Kurtsiefer, Adán Cabello, and Harald Weinfurter. Experimental demonstration of a quantum protocol for byzantine agreement and liar detection. *Phys. Rev. Lett.*, 100:070504, February 2008. doi:10.1103/PhysRevLett.100.070504.

23 Oded Goldreich, Shafi Goldwasser, and Nathan Linial. Fault-tolerant computation in the full information model. *SIAM Journal on Computing*, 27(2):506–544, 1998. doi:10.1137/S0097539793246689.

24 Mohammadtaghi Hajiaghayi, Dariusz Rafal Kowalski, and Jan Olkowski. Brief announcement: Improved consensus in quantum networks. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, pages 286–289, 2023. doi:10.1145/3583668.3594600.

25 Shang-En Huang, Seth Pettie, and Leqi Zhu. Byzantine agreement in polynomial time with near-optimal resilience. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, STOC 2022, pages 502–514, New York, NY, USA, 2022. ACM. doi:10.1145/3519935.3520015.

26 Shang-En Huang, Seth Pettie, and Leqi Zhu. Byzantine agreement with optimal resilience via statistical fraud detection. *Journal of the ACM*, 71(2):12:1–12:37, 2024. doi:10.1145/3639454.

27 Bruce M. Kapron, David Kempe, Valerie King, Jared Saia, and Vishal Sanwalani. Fast asynchronous byzantine agreement and leader election with full information. *ACM Trans. Algorithms*, 6(4), September 2010. doi:10.1145/1824777.1824788.

28 Valerie King and Jared Saia. Breaking the $O(n^2)$ bit barrier: scalable byzantine agreement with an adaptive adversary. *Journal of the ACM (JACM)*, 58(4):1–24, 2011. doi:10.1145/1989727.1989732.

29 Valerie King and Jared Saia. Byzantine agreement in expected polynomial time. *J. ACM*, 63(2), March 2016. doi:10.1145/2837019.

30 Qing-bin Luo, Kai-yuan Feng, and Ming-hui Zheng. Quantum multi-valued byzantine agreement based on d-dimensional entangled states. *International Journal of Theoretical Physics*, 58(12):4025–4032, 2019. doi:10.1007/s10773-019-04269-3.

31 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information (10th Anniversary edition)*. Cambridge University Press, 2016. URL: https://www.cambridge.org/de/academic/subjects/physics/quantum-physics-quantum-information-and-quantum-computation/quantum-computation-and-quantum-information-10th-anniversary-edition?format=HB.

32 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, April 1980. doi:10.1145/322186.322188.

33 Ramij Rahaman, Marcin Wieśniak, and Marek Żukowski. Quantum byzantine agreement via hardy correlations and entanglement swapping. *Phys. Rev. A*, 92:042302, October 2015. doi:10.1103/PhysRevA.92.042302.

34 Alexander Russell and David Zuckerman. Perfect information leader election in $\log^* n + O(1)$ rounds. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 576–583. IEEE Computer Society, 1998. doi:10.1109/SFCS.1998.743508.

35 Sam Toueg. Randomized byzantine agreements. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 163–178, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800222.806744.

A Proofs of helper lemmas

A.1 Proof of Lemma 1

Proof.

1. Since U is a permutation unitary, there exists a permutation π such that $U\left|i\right\rangle = \left|\pi(i)\right\rangle$ for any computational basis $\left|i\right\rangle$ in $\mathcal{H}$. Given any density matrix $\rho = \sum_{i,j}\rho_{i,j}\left|i\right\rangle\left\langle j\right|$ in $\mathcal{H}$, we have

$$\begin{aligned}\mathcal{M}(U\rho U^\dagger) &= \mathcal{M}\left(\sum_{i,j}\rho_{i,j}\left|\pi(i)\right\rangle\left\langle\pi(j)\right|\right)\\ &= \sum_k \left|k\right\rangle\left\langle k\right|\sum_{i,j}\rho_{i,j}\left|\pi(i)\right\rangle\left\langle\pi(j)\right|\left|k\right\rangle\left\langle k\right| = \sum_k \rho_{k,k}\left|\pi(k)\right\rangle\left\langle\pi(k)\right|,\\ U\mathcal{M}(\rho)U^\dagger &= U\sum_k\left|k\right\rangle\left\langle k\right|\sum_{i,j}\rho_{i,j}\left|i\right\rangle\left\langle j\right|\left|k\right\rangle\left\langle k\right|U^\dagger\\ &= U\sum_k\rho_{k,k}\left|k\right\rangle\left\langle k\right|U^\dagger = \sum_k\rho_{k,k}\left|\pi(k)\right\rangle\left\langle\pi(k)\right|.\end{aligned}$$

Thus $\mathcal{M}(U\rho U^\dagger) = U\mathcal{M}(\rho)U^\dagger$.

2. Since Π is an orthogonal projector in the computational basis, we have $\Pi = \sum_{i\in S}\left|i\right\rangle\left\langle i\right|$ for some set S. For any state $\rho \in \mathcal{H}$, one can verify that $\mathcal{M}(\Pi\rho\Pi^\dagger) = \Pi\mathcal{M}(\rho)\Pi^\dagger$. ◀

A.2 Proof of Lemma 2

Proof. Proof by induction on the number of messages. Initially, G and B are independent, so the state of GB is $\left|\phi_0\right\rangle_{\mathsf{G}} \otimes \left|\psi_0\right\rangle_{\mathsf{B}}$. Assuming currently the state of GB is $\sum_m \alpha_m\left|m,\phi_m\right\rangle_{\mathsf{G}} \otimes \left|\psi_m\right\rangle_{\mathsf{B}}$, consider the next message. First G and B apply a local unitary $U_{\mathsf{G}} \otimes U_{\mathsf{B}}$ to generate messages. Note that U_{G} will not change the previous transcripts $\left|m\right\rangle$. Then the state becomes

$$\sum_m \alpha_m U_{\mathsf{G}}\left|m,\phi_m\right\rangle_{\mathsf{G}} \otimes U_{\mathsf{B}}\left|\psi_m\right\rangle_{\mathsf{B}} = \sum_m \alpha_m\left|m,\phi'_m\right\rangle_{\mathsf{G}} \otimes \left|\psi'_m\right\rangle_{\mathsf{B}}.$$

- If the message is sent by G, then the system can be written as

$$\sum_m \alpha_m \sum_{m'}\beta_{m'}\left|m,m',m',\phi'_{m,m'}\right\rangle_{\mathsf{G}} \otimes \left|\psi'_m\right\rangle_{\mathsf{B}}$$

where the second m' is the message to be sent to B and the first m' is a copy to be kept by G. After sending the message, the system becomes

$$\sum_{m,m'}\alpha_m\beta_{m'}\left|m,m',\phi'_{m,m'}\right\rangle_{\mathsf{G}} \otimes \left|m',\psi'_m\right\rangle_{\mathsf{B}}.$$

- If the message is sent by B, then the system can be written as

$$\sum_m \alpha_m\left|m,\phi_m\right\rangle_{\mathsf{G}} \otimes \sum_{m'}\beta_{m'}\left|m',\psi'_{m,m'}\right\rangle_{\mathsf{B}}.$$

After sending the message, the system becomes

$$\sum_{m,m'}\alpha_m\beta_{m'}\left|m,m',\phi_m\right\rangle_{\mathsf{G}} \otimes \left|\psi'_{m,m'}\right\rangle_{\mathsf{B}}.$$ ◀

B Proof of Lemma 5

Proof. Prove by induction on k. When $k=0$, the execution $\mathcal{E}_0 = r_A$ occurs in $(\mathcal{P}_Q, \mathcal{A}_Q)$ and $(\mathcal{P}_C, \mathcal{A}_C)$ both with the probability of r_A. Assume the lemma holds for $k-1$. Consider a k-round execution $\mathcal{E}_k := r_A, (b_1, d_1), (b_2, d_2), \ldots, (b_{k-1}, d_{k-1}), (b_k, d_k)$. By inductive hypothesis, the $(k-1)$-round prefix $\mathcal{E}_{k-1} := r_A, (b_1, d_1), (b_2, d_2), \ldots, (b_{k-1}, d_{k-1})$ occurs with probability p in both $(\mathcal{P}_Q, \mathcal{A}_Q)$ and $(\mathcal{P}_C, \mathcal{A}_C)$, and the state before round k is $|Q_{k-1}\rangle := \sum_u \alpha_u |u\rangle$ and $C_{k-1} := \sum_u |\alpha_u|^2 |u\rangle\langle u|$ for $(\mathcal{P}_Q, \mathcal{A}_Q)$ and $(\mathcal{P}_C, \mathcal{A}_C)$ respectively.

Round k of $(\mathcal{P}_Q, \mathcal{A}_Q)$. After $\mathcal{A}_Q$'s action and players receiving messages, the state of the system becomes $V_k |Q_{k-1}\rangle$. Then good players first prepare a superposition state $|r_k\rangle$ of randomness in a new register R_k and prepare $|0\rangle$ in a new register A_k. Note that here $\mathsf{R}_k := (\mathsf{R}_k^{(1)}, \ldots, \mathsf{R}_k^{(n)})$, $\mathsf{A}_k := (\mathsf{A}_k^{(1)}, \ldots, \mathsf{A}_k^{(n)})$ and all other notations without superscript are defined similarly.

Then the players apply the unitary operator $U_P := \otimes_{i=1}^n U_P^{(i)}$ followed by a measurement which outputs (b_k, d_k). The measurement can be viewed as an orthogonal projector Π_{b_k, d_k} in computational basis that projects the quantum state of registers $(\mathsf{B}_k, \mathsf{D}_k)$ into values (b_k, d_k). Then the state after round k becomes

$$|Q_k\rangle := \frac{1}{\sqrt{\beta}} \Pi_{b_k, d_k} U_P \left(V_k |Q_{k-1}\rangle \otimes |0\rangle_{\mathsf{A}} \otimes |r_k\rangle_{\mathsf{R}} \right)$$

where β is the probability of getting measurement outcome (b_k, d_k).

Round k of $(\mathcal{P}_C, \mathcal{A}_C)$. The first observation is that C_k can be viewed as first applying the same operation as $(\mathcal{P}_Q, \mathcal{A}_Q)$ and then applying computational basis measurement $\mathcal{M}$ on the whole system:

$$C_k := \mathcal{M} \left(\frac{1}{\beta'} \Pi_{b_k, d_k} U_P V_k \left(C_{k-1} \otimes |0, r_k\rangle \langle 0, r_k|_{\mathsf{AR}} \right) V_k^\dagger U_P^\dagger \Pi_{b_k, d_k}^\dagger \right)$$

where β' is the probability of getting measurement outcome (b_k, d_k). The second observation is that $C_{k-1} = \mathcal{M}'(|Q_{k-1}\rangle \langle Q_{k-1}|)$ where $\mathcal{M}'$ denotes the computational basis measurement in $|Q_{k-1}\rangle$'s space. Then

$$\begin{aligned} C_k &= \mathcal{M} \left(\frac{1}{\beta'} \Pi_{b_k, d_k} U_P V_k \left(\mathcal{M}' \left(|Q_{k-1}\rangle \langle Q_{k-1}| \right) \otimes |0, r_k\rangle \langle 0, r_k|_{\mathsf{AR}} \right) V_k^\dagger U_P^\dagger \Pi_{b_k, d_k}^\dagger \right) \\ &= \frac{1}{\beta'} \Pi_{b_k, d_k} U_P V_k \mathcal{M} \left(\mathcal{M}' \left(|Q_{k-1}\rangle \langle Q_{k-1}| \right) \otimes |0, r_k\rangle \langle 0, r_k|_{\mathsf{AR}} \right) V_k^\dagger U_P^\dagger \Pi_{b_k, d_k}^\dagger. \end{aligned}$$

The second equality is because U_P, V_k are all permutation unitaries and Π_{b_k, d_k} is an orthogonal projector in computational basis, which all commute with $\mathcal{M}$ by Lemma 1. Since $\mathcal{M}$ measures a larger space than $\mathcal{M}'$, $\mathcal{M}'$ can be absorbed into $\mathcal{M}$, i.e., $\mathcal{M}\mathcal{M}' \equiv \mathcal{M}$. Thus

$$\begin{aligned} C_k &= \frac{1}{\beta'} \Pi_{b_k, d_k} U_P V_k \mathcal{M} \left(|Q_{k-1}\rangle \langle Q_{k-1}| \otimes |0, r_k\rangle \langle 0, r_k|_{\mathsf{AR}} \right) V_k^\dagger U_P^\dagger \Pi_{b_k, d_k}^\dagger \\ &= \mathcal{M} \left(\frac{1}{\beta'} \Pi_{b_k, d_k} U_P V_k \left(|Q_{k-1}\rangle \langle Q_{k-1}| \otimes |0, r_k\rangle \langle 0, r_k|_{\mathsf{AR}} \right) V_k^\dagger U_P^\dagger \Pi_{b_k, d_k}^\dagger \right) \\ &= \frac{\beta}{\beta'} \mathcal{M} \left(|Q_k\rangle \langle Q_k| \right). \end{aligned}$$

Finally, we have $\beta = \beta'$ because C_k has trace 1. Thus $C_k = \mathcal{M}\left(|Q_k\rangle \langle Q_k|\right)$ and the probability of the $\mathcal{E}_k$ occurring is $p\beta$ for both $(\mathcal{P}_Q, \mathcal{A}_Q)$ and $(\mathcal{P}_C, \mathcal{A}_C)$. ◀

C Proof of Lemma 8

Proof. Prove by induction on k. The base case $k=0$ is trivial. Assume the proposition holds for $k-1$. Consider a k-round execution $\mathcal{E}_k := r_A, (a_1, a'_1, b_1, d_1), \ldots, (a_k, a'_k, b_k, d_k)$. By inductive hypothesis, the $(k-1)$-round prefix $\mathcal{E}_{k-1} := r_A, (a_1, a'_1, b_1, d_1), \ldots, (a_{k-1}, a'_{k-1}, b_{k-1}, d_{k-1})$ occurs with probability p in both $(\mathcal{P}_Q, \mathcal{A}_Q)$ and $(\mathcal{P}_C, \mathcal{A}_C)$, and the state before round k is $|Q_{k-1}\rangle$ and $C_{k-1} := \mathcal{M}_{S_{k-1}}(|Q_{k-1}\rangle\langle Q_{k-1}|)$ for $(\mathcal{P}_Q, \mathcal{A}_Q)$ and $(\mathcal{P}_C, \mathcal{A}_C)$ respectively. Since good players' messages are fully determined by their local variables, $\mathcal{M}_{S_{k-1}}$ can be restricted to a measurement $\mathcal{M}'_{S_{k-1}}$ which does not measure the messages $\bar{S}_{k-1}$ are about to send out, i.e., $\mathsf{M}_k^{(\bar{S}_{k-1},*)}$. Then $C_{k-1} := \mathcal{M}'_{S_{k-1}}(|Q_{k-1}\rangle\langle Q_{k-1}|)$. In the following, we consider the evolution of $|Q_{k-1}\rangle$ and C_{k-1} in round k.

Step 1 and 2 of the adversary. Both $\mathcal{A}_Q$ and $\mathcal{A}_C$ apply U_k and $\mathcal{M}_k$ on S_{k-1} along with the messages $\mathsf{M}_{k-1}^{(*,S_{k-1})}$ sent to them. Since we know $\mathcal{M}_k$ will output a_k, it can be viewed as an orthogonal projector Π_{a_k} that projects into the space of a_k. Since Π_{a_k} and U_k act only on S_{k-1}'s registers and the messages $\bar{S}_{k-1}$ will send to S_{k-1}, they commute with $\mathcal{M}'_{\bar{S}_{k-1}}$. Thus the states of $(\mathcal{P}_Q, \mathcal{A}_Q)$ and $(\mathcal{P}_C, \mathcal{A}_C)$ become

$$
\begin{aligned}
|Q_{k-0.5}\rangle &:= \frac{1}{\sqrt{\gamma}}\Pi_{a_k}U_k|Q_{k-1}\rangle, \text{ and}\\
C_{k-0.5} &:= \frac{1}{\gamma'}\Pi_{a_k}U_kC_{k-1}U_k^\dagger\Pi_{a_k}^\dagger = \frac{1}{\gamma'}\Pi_{a_k}U_k\mathcal{M}'_{\bar{S}_{k-1}}(|Q_{k-1}\rangle\langle Q_{k-1}|)U_k^\dagger\Pi_{a_k}^\dagger\\
&= \mathcal{M}'_{\bar{S}_{k-1}}\left(\frac{1}{\gamma'}\Pi_{a_k}U_k|Q_{k-1}\rangle\langle Q_{k-1}|U_k^\dagger\Pi_{a_k}^\dagger\right) = \frac{\gamma}{\gamma'}\mathcal{M}'_{\bar{S}_{k-1}}(|Q_{k-0.5}\rangle\langle Q_{k-0.5}|).
\end{aligned}
$$

where γ and γ' are probabilites of $|Q_{k-1}\rangle$ and C_{k-1} outputting a_k. Since $C_{k-0.5}$ has trace 1, we have $\gamma = \gamma'$ and thus $C_{k-0.5} = \mathcal{M}'_{\bar{S}_{k-1}}(|Q_{k-0.5}\rangle\langle Q_{k-0.5}|)$.

Step 3 of the adversary. Both $\mathcal{A}_Q$ and $\mathcal{A}_C$ choose an enlarged set S_k of corrupted players. This step does not affect the state $|Q_{k-0.5}\rangle$ of $(\mathcal{P}_Q, \mathcal{A}_Q)$. By Lemma 2, we have $|Q_{k-0.5}\rangle = \sum_m \alpha_m |m, \phi_m\rangle_{\bar{S}_k}|\psi_m\rangle_{S_k}$. Since $\mathcal{M}'_{\bar{S}_{k-1}}$ can be decomposed as $\mathcal{M}'_{\bar{S}_{k-1}} \equiv \mathcal{M}_1 \otimes \mathcal{M}_2 \otimes \mathcal{M}_3$, where $\mathcal{M}_1$ acts on transcript $|m\rangle$, $\mathcal{M}_2$ acts on registers of $\bar{S}_{k-1}$ besides $|m\rangle$, and $\mathcal{M}_3$ acts on registers newly corrupted players $S_k \setminus S_{k-1}$, we have

$$
C_{k-0.5} = \mathcal{M}'_{\bar{S}_{k-1}}(|Q_{k-0.5}\rangle\langle Q_{k-0.5}|) = \sum_m |\alpha_m|^2 |m\rangle\langle m| \otimes \mathcal{M}_2(|\phi_m\rangle\langle\phi_m|) \otimes \mathcal{M}_3(|\psi_m\rangle\langle\psi_m|).
$$

In step 3 of $\mathcal{A}_C$, $\mathcal{A}_C$ has recorded the transcript m and will discard the old state $\mathcal{M}_3(|\psi_m\rangle\langle\psi_m|)$ and simulate a new state $|\psi_m\rangle$. After that, the state of $(\mathcal{P}_C, \mathcal{A}_C)$ becomes

$$
C'_{k-0.5} := \sum_m |\alpha_m|^2 |m\rangle\langle m| \otimes \mathcal{M}_2(|\phi_m\rangle\langle\phi_m|) \otimes |\psi_m\rangle\langle\psi_m| = \mathcal{M}'_{\bar{S}_k}(|Q_{k-0.5}\rangle\langle Q_{k-0.5}|).
$$

Note that we use $\mathcal{M}'_{\bar{S}_k}$ to distinguish from operator $\mathcal{M}_{\bar{S}_k}$ which also measures the newly appended registers $\mathsf{A}_k, \mathsf{R}_k$, and $\mathsf{M}_{k-1}^{(S_k,\bar{S}_k)}$ of $\bar{S}_k$ at round k.

Step 4 of the adversary and good players' action. Step 4 of $\mathcal{A}_Q$ applies U'_k followed by measurement $\mathcal{M}'_k$ which outputs a'_k on S_k's registers. The measurement can be viewed as a projector $\Pi_{a'_k}$ that projects into the space of a'_k. Then good players apply unitary U_P and projector Π_{b_k,d_k} which projects the state of registers $(\mathsf{B}_k, \mathsf{D}_k)$ into values (b_k, d_k). Thus the state of $(\mathcal{P}_Q, \mathcal{A}_Q)$ after round k becomes

$$|Q_k\rangle := \frac{1}{\sqrt{\beta}} \Pi_{b_k,d_k} U_P \left(\Pi_{a'_k} U'_k \left| Q_{k-0.5} \right\rangle \otimes |0\rangle_{\mathsf{A}} \otimes |r_k\rangle_{\mathsf{R}} \right).$$

where β is the probability of outputting a'_k, b_k, and d_k.

Step 4 of $\mathcal{A}_C$ will additionally apply a measurement $\mathcal{M}_{msg}$ on messages $\mathsf{M}_{k-1}^{(S_k,\bar{S}_k)}$ sent from S_k to $\bar{S}_k$. The good players' action of $\mathcal{P}_C$ can be viewed as applying the same operation as $\mathcal{P}_Q$ and then measuring good players $\bar{S}_k$'s space in computational basis. Thus the state of $(\mathcal{P}_C, \mathcal{A}_C)$ becomes

$$\begin{aligned} C_k &:= \mathcal{M}_{\bar{S}_k} \left(\frac{1}{\beta'} \Pi_{b_k,d_k} U_P \left(\mathcal{M}_{msg} (\Pi_{a'_k} U'_k C'_{k-0.5} U'^{\dagger}_k \Pi^{\dagger}_{a'_k}) \otimes |0, r_k\rangle \langle 0, r_k|_{\mathsf{AR}} \right) U^{\dagger}_P \Pi^{\dagger}_{b_k,d_k} \right) \\ &= \mathcal{M}_{\bar{S}_k} \left(\frac{1}{\beta'} \Pi_{b_k,d_k} U_P \left(\mathcal{M}_{msg} (\Pi_{a'_k} U'_k \mathcal{M}'_{\bar{S}_k} \left(|Q_{k-0.5}\rangle \langle Q_{k-0.5}| \right) U'^{\dagger}_k \Pi^{\dagger}_{a'_k}) \otimes |0, r_k\rangle \langle 0, r_k|_{\mathsf{AR}} \right) \right. \\ &\qquad \left. U^{\dagger}_P \Pi^{\dagger}_{b_k,d_k} \right) \end{aligned}$$

where β' is the probability of outputting a'_k, b_k, and d_k. Similar to Fail-stop case, $\mathcal{M}_{\bar{S}_k}$ and $U_P \Pi_{b_k,d_k}$ commute by Lemma 1. $\mathcal{M}'_{\bar{S}_k}$ and $\Pi_{a'_k} U'_k$ also commute because they act on $\bar{S}_k$ and S_k separately. Thus

$$\begin{aligned} C_k &= \frac{1}{\beta'} \Pi_{b_k,d_k} U_P \mathcal{M}_{\bar{S}_k} \mathcal{M}_{msg} \mathcal{M}'_{\bar{S}_k} \left(\Pi_{a'_k} U'_k |Q_{k-0.5}\rangle \langle Q_{k-0.5}| U'^{\dagger}_k \Pi^{\dagger}_{a'_k} \otimes |0, r_k\rangle \langle 0, r_k|_{\mathsf{AR}} \right) U^{\dagger}_P \Pi^{\dagger}_{b_k,d_k} \\ &= \frac{1}{\beta'} \Pi_{b_k,d_k} U_P \left(\mathcal{M}_{\bar{S}_k} (\Pi_{a'_k} U'_k |Q_{k-0.5}\rangle \langle Q_{k-0.5}| U'^{\dagger}_k \Pi^{\dagger}_{a'_k}) \otimes |0, r_k\rangle \langle 0, r_k|_{\mathsf{AR}} \right) U^{\dagger}_P \Pi^{\dagger}_{b_k,d_k} \\ &= \mathcal{M}_{\bar{S}_k} \left(\frac{1}{\beta'} \Pi_{b_k,d_k} U_P \left(\Pi_{a'_k} U'_k |Q_{k-0.5}\rangle \langle Q_{k-0.5}| U'^{\dagger}_k \Pi^{\dagger}_{a'_k} \otimes |0, r_k\rangle \langle 0, r_k|_{\mathsf{AR}} \right) U^{\dagger}_P \Pi^{\dagger}_{b_k,d_k} \right) \\ &= \frac{\beta}{\beta'} \mathcal{M}_{\bar{S}_k} \left(|Q_k\rangle \langle Q_k| \right). \end{aligned}$$

where the second equality is because $\mathcal{M}_{\bar{S}_k}$ measures a larger space than $\mathcal{M}_{msg} \mathcal{M}'_{\bar{S}_k}$, thus $\mathcal{M}_{\bar{S}_k} \mathcal{M}_{msg} \mathcal{M}'_{\bar{S}_k} \equiv \mathcal{M}_{\bar{S}_k}$.

Finally, since C_k has trace 1, we have $\beta = \beta'$ and thus $C_k = \mathcal{M}_{\bar{S}_k} \left(|Q_k\rangle \langle Q_k| \right)$. The probability of $\mathcal{E}_k$ occurring is $p\gamma\beta$ for both cases. ◀

Communication Requirements for Linearizable Registers

Raïssa Nataf
Technion, Haifa, Israel

Yoram Moses
Technion, Haifa, Israel

Abstract

While linearizability is a fundamental correctness condition for distributed systems, ensuring the linearizability of implementations can be quite complex. An essential aspect of linearizable implementations of concurrent objects is the need to preserve the real-time order of operations. In many settings, however, processes cannot determine the precise timing and relative real-time ordering of operations. Indeed, in an asynchronous system, the only ordering information available to them is based on the fact that sending a message precedes its delivery. We show that as a result, message chains must be used extensively to ensure linearizability. This paper studies the communication requirements of linearizable implementations of atomic registers in asynchronous message passing systems. We start by proving two general theorems that relate message chains to the ability to delay and reorder actions and operations in an execution of an asynchronous system, without the changes being noticeable to the processes. These are then used to prove that linearizable register implementations must create extensive message chains among operations of *all* types. In particular, our results imply that linearizable implementations in asynchronous systems are necessarily costly and nontrivial, and provide insight into their structure.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases linearizability, atomic registers, asynchrony, message chains, real time

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.33

Funding Yoram Moses is the Israel Pollak academic chair at the Technion. This work was supported in part by the Israel Science Foundation under grant ISF 2061/19.

Acknowledgements We thank Gal Assa, Naama Ben David, and an anonymous referee for very useful comments that improved the presentation of this paper. We alone are responsible for any errors or misrepresentations.

1 Introduction

Linearizability [15] is a fundamental correctness criterion and is the gold standard for concurrent implementations of shared objects. Informally, an object implementation is linearizable if in each one of its executions, operations appear to occur instantaneously, in a way that is consistent with the execution and the object's specification. Linearizable implementations have been developed for a variety of concurrent objects [1, 20, 14] and is also widely used in the context of *state-machine replication* (SMR) mechanisms [23, 11, 22]. Understanding the costs that linearizable implementations imply and optimizing their performance is thus crucial. Lower bounds on linearizable implementations are rare in the literature. Our paper makes a significant step towards capturing inherent costs of linearizability in the important case of linearizable register implementations, and provides a new formal tool for capturing the necessary structure of communication in register implementations.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 33; pp. 33:1–33:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In an execution of a linearizable implementation, the actions performed and values observed by processes depend on the real-time ordering of non-overlapping operations [15]. However, processes do not have direct access to real time in the asynchronous setting, and this makes satisfying linearizability especially challenging. The only way processes can obtain information about the real-time order of events in asynchronous message-passing systems is via *message chains* (*cf.* Lamport's *happens before* relation [18]). Roughly speaking, a message chain connects process i at (real) time t and process $j \neq i$ at t' if there is a sequence of messages starting with a message sent by i at or after t, ending with a message received by j no later than time t', such that every message is received by the sender of the following message in the sequence, before the following message is sent.[1] Message chains can be used to ensure the relative real-time order of events. Moreover, as we formally show, in the absence of a message chain relating events at distinct processes, there can be no way to tell what their real-time order is. This paper establishes the central role that message chains must play in achieving linearizability in an asynchronous system.

Registers constitute a central abstraction in distributed computing. In their seminal paper [4], Attiya, Bar-Noy and Dolev provide a linearizable implementation of single-writer multi-reader (SWMR) registers in an asynchronous message passing model where processes are prone to crash failures. This implementation was extended to the multi-writer multi-reader (MWMR) case in [19]. Since then, there has been significant interest in implementing registers in asynchronous message passing models. In [4], quorum systems are used to guarantee a message chain between *every pair* of non-overlapping operations. This is costly, of course, both in communication and in execution time. Is it necessary?

In a linearizable implementation of a MWMR register, every process can issue reads and writes, and a read should intuitively return the most recent value written. It is to be expected that a reader must be able to access previous write operations, and especially the one whose value its read operation returns. But should writing a new value, for example, require message chains from all previous reads and writes? Must a process that has read a value communicate this fact to others? Interestingly, we show in this work that typically, the answer is *yes*. Moreover, we prove that every operation of a *fault-tolerant* implementation of a MWMR register must communicate with a quorum set before it completes.

The main contributions of this paper are

1. We show that in a linearizable implementation of a register in an asynchronous setting, every operation, regardless of type, might need to have a message chain to arbitrary operations in the future. Moreover, in an f-resilient implementation, before a process can complete an operation, it must construct a round-trip message chain interaction with nodes in a quorum set of size greater than f. These requirements apply to *every* execution and thus, provide a natural way for establishing lower bounds on the performance of register implementations and related applications not only in the worst case, but also in optimistic executions (a.k.a. fast paths) [22, 17, 10]. We expect this work to serve as a tool for analyzing the efficiencies of existing implementations and also as a guide for implementing new linearizable objects in the future.
2. We show these results by formulating and proving two useful and general theorems about coordination in asynchronous systems. One relates message chains to the ability to delay particular actions in an execution of an asynchronous system for an arbitrary amount of time, without the delay being noticeable to any process in the system. The other relates them to the ability to change the relative real-time order of operations on concurrent objects in manners that may cause violations of linearizability requirements.

[1] A formal definition appears in Section 3.2.

Interestingly, a significant amount of communication in a linearizable implementation is required for timing purposes, rather than for transferring information about data values. Our results apply verbatim if message passing is replaced by communication via asynchronous single-writer single-reader (SWSR) registers or in hybrid models ([2]) for a suitably modified notion of message chains. They also extend to other variants of linearizability, such as strict linearizability [3].

This paper is structured as follows: Section 2 presents related work. In Section 3 we present the model and preliminary definitions and results about message chains, real time ordering and the local equivalence of runs. In Section 4 we prove a theorem about the ability to delaying actions in a way that processes cannot notice. This is used in Section 5 to show that certain operations can be reordered in a run, in a similar fashion. These results, which can be applied to arbitrary objects, are next used for the study of atomic register implementations. Section 6 contains definitions of registers and linearization in our setting. Section 7 provides general results showing the need for message chains between operations in executions of linearizable register implementations. In Section 8 we show how the presence of failures combined with the results of the previous sections imply the necessity of using quorum systems.

2 Related Work

Attiya, Bar-Noy and Dolev's paper (ABD) [4] shows how to implement shared memory via message passing in an asynchronous message passing model where processes are prone to crash failures. Their algorithm (which we shall call ABD) is f-resilient and makes use of quorum systems. Each write or read operation performs two communication rounds. In each communication round by p, process p sends messages to all n processes and waits for replies from $n - f$ processes before it proceeds to the next communication round.

In [10] and [16], Dutta et al. and Huang et al., respectively, consider a model consisting of disjoint sets of servers, writers and readers and where at least one process can fail ($f \geq 1$). They study implementations of an atomic register where read or write operations are *fast*, by which they mean that the operations terminate following a single communication round. In [10], an SWMR register implementation is provided with fast reads and writes, if the number of readers is small enough relative to the number of servers and the maximal number of failures. They also prove that MWMR register implementations with both fast read and fast writes are impossible. [16] proves that implementations with fast writes are impossible and by showing under which conditions (on the number of failures) implementations with fast reads exist. The models of [10, 16] assume crash failures. Our results in Sections 4-7 are valid both when processes are guaranteed to be reliable (no failures) and in the presence of crash failures.

In [21] Naser-Pastoriza et al. consider networks where channels may disconnect. As one of that paper's main contributions, it establishes minimal connectivity requirements for linearizable implementations of registers in a crash fault environment where channels can drop messages. Informally, it is shown that (1) all processes where obstruction-freedom holds must be strongly connected via correct channels; and (2). If the implementation tolerates k process crashes and $n = 2k + 1$, then any process where obstruction-freedom holds must belong to a set of more than k correct processes strongly connected by correct channels.

The works [6, 8, 9] show that *quorum* failure detectors, introduced by Delporte-Gallet et al. in [8], are the weakest failure detectors enabling the implementation of an atomic register object in asynchronous message passing systems. This class of failure detectors capture the minimal information regarding failures that processes must possess to in linearizable implementation of registers.

Variants of linearizability that differ in the way crashes are handled have been defined in the context of NVRAMs; see Ben-David *et. al* [5] for a survey. Another important variant is *strong* linearizability, introduced by Golab *et. al* in [13]. There it is shown that in a randomized algorithm, executions behave exactly as if using atomic objects if and only if the implementation is strongly linearizable. It has been shown that registers do not have strongly linearizable implementations in the message passing model. Our results in Sections 4–6 are valid in asynchronous models in general and can thus also be used in the analysis and the study of such variants of linearizability.

3 Model and Preliminary Definitions

3.1 Model

While asynchronous systems are often captured using an interleaving model, we adopt the asynchronous message passing model from [12], in which several events can take places at the same time. This facilitates reasoning about the time at which actions and operations occur, and analyzing the possibility of modifying the timing of some operations while leaving the timing of other operations unchanged. We briefly describe the model here and refer the reader to Appendix A for the complete detailed model. The detailed model is required mainly for the proof of Theorem 6 which is lays the technical basis for most of our analysis.

We consider an asynchronous message passing model with n processes, connected by a communicated network, modelled by a directed graph where an edge from process i to process j is called a *channel*, and denoted by $\mathsf{chan}_{i,j}$. The environment, which plays the role of the adversary, is in charge of scheduling processes, of delivering messages, and of invoking operations (such as reads, writes etc.) at a process. A run of the system is an infinite sequence $r = r(0), r(1), \ldots$ of global states, where each global state $r(m)$ determines a local state for each process, denoted by $r_i(m)$. We identify time with the natural numbers, and consider $r(m)$ to be the system's state at time m in r. For ease of exposition, we assume that messages along a channel are delivered in FIFO order. Moreover, we assume that the local state of a process i keeps track of of the events it has been involved in so far: all actions it has performed, all messages it sent and received, and all operations invoked at i, up to the current time. Asynchrony of the system is captured by assuming that processes moves, message deliveries and operation invocation are scheduled in an arbitrary nondeterministic order. Thus messages can take any amount of time to be delivered, and processes can refrain from performing moves for arbitrarily long time intervals. We consider actions to be performed in *rounds*, where round m occurs between time m and time $m+1$. The transition from $r(m)$ to $r(m+1)$ is based on the actions performed by the environment and by all processes that move in round $m+1$.

A process i is said to be correct in r if it is allowed to move (by the environment) infinitely often in r. Otherwise process i is *faulty* (or *crashes*) in r. We say that a message μ is *lost* in r if it is sent in r and never delivered. A system is said to be *reliable* if no process ever fails and no message is ever lost, in any of its runs. Finally, a protocol is said to be f*-resilient* if it acts correctly in all runs in which no more than f processes are faulty.

3.2 Message Chains, Real-time Ordering and Local Equivalence

As stated in the introduction, the real-time order of events in a system plays a central role in linearizable protocols. The main source of information about the order of events in asynchronous systems are message chains. We denote by $\theta = \langle p, t\rangle$ a **process-time** pair (or a *node*) consisting of the process p and time t. Such a pair is used to refer to the point on p's timeline at real time t. We can inductively define a message chain between nodes of a given run as follows.

▶ **Definition 1** (Message chains). *There is* **a message chain** *from* $\theta = \langle p, t\rangle$ *to* $\theta' = \langle q, t'\rangle$ *in a run* r*, denoted by* $\theta \rightsquigarrow_r \theta'$*, if*

(1a) $p = q$ *and* $t < t'$*,*

(1b) p *sends a message to* q *in round* $t+1$ *of* r*, which arrives no later than in round* t'*, or*

(2) *there exists* θ'' *such that* $\theta \rightsquigarrow_r \theta''$ *and* $\theta'' \rightsquigarrow_r \theta'$*.*

Lamport calls "$\rightsquigarrow_r$" the *happens before* relation [18]. As we now show, the existence of message chains indeed implies real-time ordering. We write $\theta <_r \theta'$ if $\theta = \langle p, t\rangle$ and $\theta' = \{q, t'\}$ are nodes in r and $t < t'$. An immediate implication of Definition 1 is

▶ **Observation 2.** *If* $\theta \rightsquigarrow_r \theta'$ *then* $\theta <_r \theta'$*.*

Proof. Let $\theta = \langle p, t\rangle$ and $\theta' = \langle q, t'\rangle$. The proof is by induction on the minimal number of applications of step (2) in Definition 1 needed to establish that $\theta \rightsquigarrow_r \theta'$. If $\theta \rightsquigarrow_r \theta'$ by (1a) then $t < t'$. Similarly, if it is by (1b), then $t < t'$ because a message sent in round $t+1$ can only arrive in a round $t' \geq t+1 > t$. Finally, if $\theta \rightsquigarrow_r \theta'$ by clause (2), then for some node $\theta'' = \langle p'', t''\rangle$ we have that $\theta \rightsquigarrow_r \theta''$ and $\theta'' \rightsquigarrow_r \theta'$, where, inductively, $t < t''$ and $t'' < t$. It follows that $t < t'$, as required. ◀

The converse is not true: It is possible for θ to appear before θ' in real time, without a message chain between them. As we shall see, however, in the absence of a message chain, processes will not be able to detect the ordering between the nodes.

Roughly speaking, the information available to a process at a given point is determined by its local state there. A process is unable to distinguish between runs in which it passes through the same sequence of local states. We will find it useful to consider when two runs cannot *ever* be distinguished by any of the processes. Formally:

▶ **Definition 3** (Local Equivalence). *Two runs* r *and* r' *are called* locally equivalent*, denoted by* $r \approx r'$*, if for every process* j*, a local state* ℓ_j *of* j *appears in* r *iff* ℓ_j *appears in* r'*.*

Recall that the local state of a process i consists of its local history so far. Consequently, an equivalent definition of local equivalence is that if two runs are locally equivalent, then every process starts in the same state, performs the same actions and sends and receives the same messages, all in the same order, in both runs.

A node $\theta = \langle i, t\rangle$ of i in r is said to *correspond* to node $\theta' = \langle j, t'\rangle$ of r', denoted by $\theta \sim \theta'$, if $i = j$ (they refer to the same process) and the process has the same local state at both (i.e., $r_i(r) = r'_i(t')$). We will make use of the following properties of local equivalence (the proof of Lemma 4 appears in the Appendix):

▶ **Lemma 4.** *Let* r *and* r' *be two runs such that* $r \approx r'$*. Then*

(i) *If* $\theta_1 \rightsquigarrow_r \theta_2$ *then* $\theta'_1 \rightsquigarrow_{r'} \theta'_2$ *holds for all nodes* θ'_1 *and* θ'_2 *of* r' *such that* $\theta_1 \sim \theta'_1$ *and* $\theta_2 \sim \theta'_2$

(ii) *If* r *is a run of protocol* P*, then* r' *is also a run of* P

(iii) *A process* i *fails in* r *iff it fails in* r'*, and similarly*

(iv) *A message* μ *is lost in* r *iff the same message is lost in* r'

4 Delaying the Future while Maintaining the Past

We are now ready to state and prove the main theorem that will allow us to capture the subtle interaction between message chains and the ability to reorder operations in an asynchronous system.

▶ **Definition 5** (The past of θ). *For a node θ in a run r, we define* $\mathsf{past}_r(\theta) \triangleq \{\theta' \mid \theta' \rightsquigarrow_r \theta\}$.

Chandy and Misra have already shown that, in a precise sense, in an asynchronous system, a process at a given node cannot know about the occurrence of any events except for ones that appear in its past [7]. Our theorem will show that for any given node θ in a run r (which we think of as a "pivot node") **all** events that occur outside $\mathsf{past}_r(\theta)$ can be pushed into the future by an arbitrary amount $\Delta > 0$, without any node observing the change.

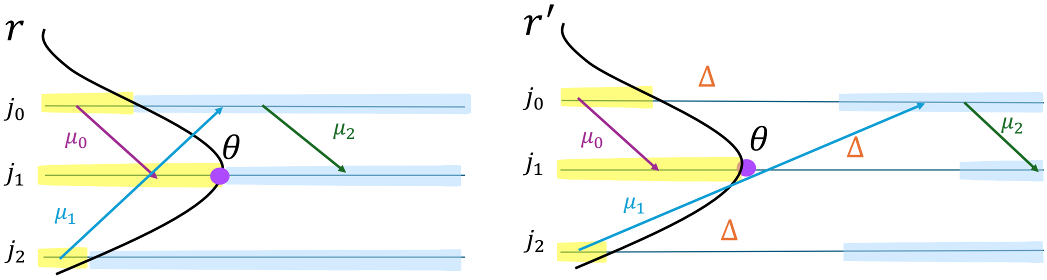


Figure 1 Delaying events by Δ relative to the past of a node θ (the "pivot").

▶ **Theorem 6** (Delaying the future). *Fix a run r of a protocol P, a node $\theta = \langle i, t\rangle$, and a delay $\Delta > 0$. For each process j denote by t_j the minimal time $l \geq 0$ such that $\langle j, l\rangle \not\rightsquigarrow_r \theta$ (i.e., $\langle j, t_j\rangle$ is the first point of j that is not in the past of θ in r). Then there exists a run $r' \approx r$ satisfying, for every process j:*

$$r_j(m) = \begin{cases} {r'}_j(m) & \textit{for all } m \leq t_j \\ {r'}_j(m+\Delta) & \textit{for all } m \geq t_j + 1 \end{cases}$$

This theorem lays the technical foundation for most of our analysis in this paper. We start by providing a sketch of its proof, and follow with the full proof.

Proof sketch. Recall that we are given r, θ and Δ. For every process j there is an earliest time t_j such that $\langle j, t_j\rangle \notin \mathsf{past}_r(\theta)$. We now construct a run r' that agrees with r on all nodes of $\mathsf{past}_r(\theta)$. I.e., for every node $\theta' = \langle p, t'\rangle \in \mathsf{past}_r(\theta)$, then the same actions occur in round t' on p's timeline, and $r_p(t') = r'_p(t')$. Moreover, outside of $\mathsf{past}_r(\theta)$ the run r' is defined as follows. The environment in r' "puts to sleep" every process j (by performing $\mathtt{skip}_j$ actions) for a duration of Δ rounds starting from round $t_j + 1$ and ending in round $t_j + \Delta$. Every message that, in r, is delivered to j at a round $m > t_j$ is delivered Δ rounds later, i.e., in round $m + \Delta$, in r'. Similarly, every message sent by i after time t_i in r is sent Δ rounds later in r'. A crucial property of this construction is that, by definition of $\rightsquigarrow_r$, if the sending of a message is delayed by Δ in r' – the sending node is *not* in $\mathsf{past}_r(\theta)$ – then its delivery is delayed by Δ as well. Consequently, every message sent in r' is delivered at a time that is greater than the time it is sent, and so r' is a legal run. What remains is to check that the run r' is indeed locally equivalent to r. This careful and somewhat tedious task is performed in the full proof that follows below. ◀

As illustrated in Figure 1, the run r' contains a band of inactivity that is Δ rounds deep in front of the boundary of $\mathsf{past}_r(\theta)$. Since Δ can be chosen arbitrarily, Theorem 6 can be used to rearrange any activity that does not involve nodes of $\mathsf{past}_r(\theta)$, even events that may be very early, to occur strictly after θ in r'. Crucially, no process is ever able to distinguish among the two runs.

Proof of Theorem 6. To simplify the case analysis in our proof, we define

$$\mathtt{shift}_\Delta[m,t_j] \;\triangleq\; \begin{cases} m & m \le t_j \\ m+\Delta & m \ge t_j+1 \end{cases}$$

Notice that the range of $\mathtt{shift}_\Delta[m,t_j]$ for $m \ge 0$ is the set of times m' not in the interval $t_j+1 \le m' \le t_j+\Delta$. Moreover, observe that $\mathtt{shift}_\Delta[m-1,t_j] = \mathtt{shift}_\Delta[m,t_j]-1$ for all $m>0$ such that $m \ne t_j+1$. We shall construct a run $r' \approx r$ satisfying, for every process j and all $m \ge 0$:

(i) $r_j(m) = {r'}_j(\mathtt{shift}_\Delta[m,t_j])$ for all $m \ge 0$, and

(ii) Process j performs the same actions and receives the same messages in round m of r and in round $\mathtt{shift}_\Delta[m,t_j]$ of r', for all $m \ge 1$.

We construct r' as follows. Both runs start in the same initial state: $r'(0) = r(0)$. Denote the environment's action in r in round m by $\eta(r,m) = (\eta_1(r,m),\ldots,\eta_n(r,m))$. For every process j the environment's actions η_j satisfies $\eta_j(r',m') \triangleq \mathtt{skip}_j$ for all m' in the range $t_j+1 \le m' \le t_j+\Delta$. For all $m \ge 0$ we define

$$\eta_j(r',\mathtt{shift}_\Delta[m,t_j]) \;\triangleq\; \begin{cases} \eta_j(r,m) & \text{if } \eta_j(r,m) \in \{\mathtt{skip}_j, \mathtt{move}_j, \mathtt{invoke}_j(x)\} \\ \mathtt{deliver}_j(|\mu,\mathtt{shift}_\Delta[m_h,t_h]|,h) & \text{if } \eta_j(r,m) = \mathtt{deliver}_j(|\mu,m_h|,h) \end{cases}$$

As for process actions, for all j and $m>0$, if $\eta_j(r',\mathtt{shift}_\Delta[m,t_j]) = \mathtt{move}_j$ and ${r'}_j(m-1) = r_j(m-1)$ then j performs the same action $\alpha_j \in P_j(r_j(m-1))$ in round $\mathtt{shift}_\Delta[m,t_j]$ of r' as in round m of r, and otherwise it performs an arbitrary action from $P_j({r'}_j(\mathtt{shift}_\Delta[m-1,t_j])$ in round $\mathtt{shift}_\Delta[m,t_j]$ of r'. Notice that, by definition, all processes follow the protocol $P=(P_1,\ldots,P_n)$ in r'. Moreover, observe the following useful property of r':

▷ Claim 7. ${r'}_j(\mathtt{shift}_\Delta[m,t_j]-1) = {r'}_j(\mathtt{shift}_\Delta[m-1,t_j])$ for all $m>0$.

Proof. We consider two cases:

- $m = t_j+1$: Observe that $r'(t_j+\Delta+1) = r'(t_j+\Delta) = \cdots = r'(t_j)$ since by definition of the run r', we have that $\eta_j(r',m') = \mathtt{skip}_j$ for all $t_j+1 \le m' \le t_j+\Delta$. So, ${r'}_j(\mathtt{shift}_\Delta[m,t_j]-1) = {r'}_j(\mathtt{shift}_\Delta[t_j+1,t_j]-1) = {r'}_j(t_j+1+\Delta-1) = {r'}_j(t_j+\Delta) = {r'}_j(t_j) = {r'}_j(\mathtt{shift}_\Delta[m-1,t_j])$.
- $0 < m \ne t_j+1$: If $m \le t_j$ then by definition of $\mathtt{shift}_\Delta$ we have that $\mathtt{shift}_\Delta[m,t_j] = m$ and $\mathtt{shift}_\Delta[m-1,t_j] = m-1 = \mathtt{shift}_\Delta[m,t_j]-1$. Similarly, if $m > t_j+1$ then $\mathtt{shift}_\Delta[m,t_j] = m+\Delta$ and $\mathtt{shift}_\Delta[m-1,t_j] = m-1+\Delta = \mathtt{shift}_\Delta[m,t_j]-1$. In both cases we obtain that ${r'}_j(\mathtt{shift}_\Delta[m,t_j]-1) = {r'}_j(\mathtt{shift}_\Delta[m-1,t_j])$, as desired. ◁

We are now ready to prove that r' is a legal run of P satisfying (i) and (ii). We prove this by induction on $m \ge 0$, for all processes j.

Base, $m=0$: By definition of r' we have that ${r'}_j(0) = r_j(0)$.

Step, $m>0$: Assume inductively that (i) and (ii) hold for all processes h at all times strictly smaller than m. We start by establishing:

▷ Claim 8. If a message μ sent by a process h at time m_h is delivered to j in round m of r, then $|\mu,\mathtt{shift}_\Delta[m_h,t_h]| \in \mathit{chan}_{hj}$ at time $\mathtt{shift}_\Delta[m,t_j]-1$ of r'.

Proof. Clearly, if μ is delivered to j in round m of r then $\eta_j(r,m) = \texttt{deliver}_j(|\mu, m_h|, h)$ for some process $h \neq j$ and round $m_h < m$. By the inductive assumption for h and $m_h < m$, we have that μ is sent in round $\texttt{shift}_\triangle[m_h, t_h]$ of r'. In addition, by definition of r', for all $m' < \texttt{shift}_\triangle[m, t_j]$ it holds that $\eta_j(r', m') \neq \texttt{deliver}_j(|\mu, \texttt{shift}_\triangle[m_h, t_h], h)$. So $|\mu, \texttt{shift}_\triangle[m_h, t_h]| \in \mathit{chan}_{hj}$ at time $\texttt{shift}_\triangle[m, t_j] - 1$ in r'. ◁

Recall that we have by the inductive assumption that $r'_j(\texttt{shift}_\triangle[m-1, t_j]) = r_j(m-1)$. Claim 7 thus implies that

$$r'_j(\texttt{shift}_\triangle[m, t_j] - 1) \;=\; r_j(m-1). \tag{1}$$

We can now show that (i) and (ii) hold for j and m by cases depending on the environment's action $\eta_j(r,m)$ in round m of r:

- $\eta_j(r,m) = \texttt{skip}_j$: By definition of η_j for r', we have that $\eta_j(r', \texttt{shift}_\triangle[m]) = \texttt{skip}_j$. So, $r'_j(\texttt{shift}_\triangle[m, t_j]) = r'_j(\texttt{shift}_\triangle[m, t_j] - 1) = r_j(m-1)$, proving (i). Moreover, no action is performed by j neither in r nor in r' and no message is delivered to j in either case, ensuring that (ii) also holds.
- $\eta_j(r,m) = \texttt{invoke}_j(x)$: In this case, $\eta_j(r', \texttt{shift}_\triangle[m]) = \texttt{invoke}_j(x)$, implying that $r'_j(\texttt{shift}_\triangle[m, t_j]) = r_j(m)$.
- $\eta_j(r,m) = \texttt{move}_j$: In this case, $\eta_j(r', \texttt{shift}_\triangle[m]) = \texttt{move}_j$ by definition of η_j for r'. By (1) we have that $r'_j(\texttt{shift}_\triangle[m, t_j] - 1) = r_j(m-1)$. So by definition of r', process j performs the same action $\alpha_j \in P_j(r_j(m))$ in the round $\texttt{shift}_\triangle[m, t_j]$ of r' as it does in the round m of r. This also ensures $r'_j(\texttt{shift}_\triangle[m, t_j]) = r_j(m)$. In addition, no message is delivered in round m of r and none is delivered to it in round $\texttt{shift}_\triangle[m, t_j]$ of r'.
- $\eta_j(r,m) = \texttt{deliver}_j(|\mu, m_h|, h)$: In this case, no action is performed by j. By definition, $\eta_j(r', \texttt{shift}_\triangle[m, t_j]) = \texttt{deliver}_j(|\mu, \texttt{shift}_\triangle[m_h, t_h]|, h)$. Recall that by (1) we have $r'_j(\texttt{shift}_\triangle[m, t_j] - 1) = r_j(m-1)$. We now show that μ is delivered in r in round m iff it is delivered in r' in round $\texttt{shift}_\triangle[m, t_j]$.
  - If μ is delivered in round m of r then by Claim 8 we have that $|\mu, \texttt{shift}_\triangle[m_h, t_h]| \in \mathit{chan}_{hj}$ at time $\texttt{shift}_\triangle[m, t_j] - 1$ in r' so μ is delivered in round $\texttt{shift}_\triangle[m]$ of r' as well.
  - Otherwise, i.e., μ is not delivered in round m of r. Assume by way of contradiction that μ is delivered in round $\texttt{shift}_\triangle[m, t_j]$ of r'. So $|\mu, \texttt{shift}_\triangle[m_h, t_h]| \in \mathit{chan}_{hj}$ at time $\texttt{shift}_\triangle[m, t_j] - 1$ in r' and thus μ is sent in round $\texttt{shift}_\triangle[m_h, t_h] < \texttt{shift}_\triangle[m, t_j]$ of r'. By the inductive hypothesis, μ is sent in round m_h of r . Since μ is not delivered in round m of r, while $\eta_j(r,m) = \texttt{deliver}_j(|\mu, m_h|)$, we have that μ is delivered in some round $m' < m$ of r. So by Claim 8, μ must be delivered at time $\texttt{shift}_\triangle[m', t_j] < \texttt{shift}_\triangle[m, t_j]$ in r'. Hence, $|\mu, \texttt{shift}_\triangle[m_h, t_h]| \notin \mathit{chan}_{hj}$ at time $\texttt{shift}_\triangle[m, t_j] - 1$ in r', contradicting the fact that μ is delivered in round $\texttt{shift}_\triangle[m, t_j]$ of r'.

  We thus obtain that $r_j(m) = r'_j(\texttt{shift}_\triangle[m, t_j])$, and that the same actions (none in this case) and the same messages are delivered in round m of r and in round $\texttt{shift}_\triangle[m, t_j]$ of r'. ◀

5 Operations

To capitalize on the power of Theorem 6, we now set out to show how operations on distributed objects can be rearranged while maintaining local equivalence. We consider operations that are associated with individual processes. An operation O of type O[2] starts with an invocation

[2] While processes are typically able to perform particular types of operations on concurrent objects, such as reads, writes, etc., many different instances of an operation may appear in a given run. Every instance of an operation has a type.

input $\texttt{invoke}_i(O, \text{arg})$ from the environment to process i, and ends when process i performs a matching response action $\texttt{return}_i(O, \text{arg}) \in Act_i$. Operation invocations in our model are nondeterministic and asynchronous – the environment can issue them at arbitrary times.[3] Operations can have invocation or return parameters, which appeared as arg in the above notation. E.g., a write invocation to a register will have a parameter v (the value to be written), while the response to a read on the register will provide the value v' being read.

We say that an operation $\texttt{X}$ occurs between nodes $\theta = \langle i, t\rangle$ and $\theta' = \langle i, t'\rangle$ in r if $\texttt{X}$'s invocation by the environment (of the form $\texttt{invoke}_i(\cdot)$) occurs in round t in r and process i performs $\texttt{X}$'s response action in round t'. In this case we denote $\texttt{X}.s \triangleq \theta$ and $\texttt{X}.e \triangleq \theta'$, and use $t_{\texttt{X}.s}(r)$ to denote the operation's starting time t and $t_{\texttt{X}.e}(r)$ to denote its ending time t'. When the run is clear from the context we do not precise it. An operation $\texttt{O}$ is *completed* in a run r if r contains both the invocation and response of $\texttt{O}$, otherwise $\texttt{O}$ is *pending*. Observe that in a crash prone environment, it is not possible to guarantee that every operation completes, since once a process crashes, it is not able to issue a response.

▶ **Definition 9** (Real-time order and concurrency). *For two operations* $\texttt{X}$ *and* $\texttt{Y}$ *in* r *we say that* $\texttt{X}$ precedes $\texttt{Y}$ *in* r*, denoted* $\texttt{X} <_r \texttt{Y}$*, if* $t_{\texttt{X}.e}(r) < t_{\texttt{Y}.s}(r)$*, i.e., if* $\texttt{X}$ *completes before* $\texttt{Y}$ *is invoked. If neither* $\texttt{X}$ *precedes* $\texttt{Y}$ *nor* $\texttt{Y}$ *precedes* $\texttt{X}$*, then* $\texttt{X}$ *and* $\texttt{Y}$ *are considered* concurrent *in* r*. Finally,* $\texttt{X}$ *is said to* run in isolation *in* r *if no operation is concurrent to* $\texttt{X}$ *in* r*.*

▶ **Definition 10** (Message chains among operations). *We write* $\texttt{X} \rightsquigarrow_r \texttt{Y}$ *and say that there is a message chain between the operations* $\texttt{X}$ *and* $\texttt{Y}$ *in* r *if* $\texttt{X}.s \rightsquigarrow_r \texttt{Y}.e$*.*

Notice that $\texttt{X} \rightsquigarrow_r \texttt{Y}$ does not imply that $\texttt{X}$ happens before $\texttt{Y}$ in real time (i.e., it does not imply that $\texttt{X} <_r \texttt{Y}$). Rather, it only implies that $\texttt{Y}$ does not end before $\texttt{X}$ starts (i.e., $\texttt{Y} \not<_r \texttt{X}$). Moreover, while "$\rightsquigarrow_r$" among individual nodes is transitive, "$\rightsquigarrow_r$" among operations is not.

An operation $\texttt{X}$ of i in the run r is said to *correspond* to operation $\texttt{X}'$ of j in r', denoted by $\texttt{X} \sim \texttt{X}'$, if $i = j$ (they are performed by the same process), $\texttt{X}.s \sim \texttt{X}'.s$ and $\texttt{X}.e \sim \texttt{X}'.e$. Note that for locally equivalent runs $r \approx r'$, for every operation $\texttt{X}$ in r there is a corresponding operation $\texttt{X}'$ in r'. In the sequel, we will often refer to corresponding operations in different runs by the same name. Observe that, by the definition of $\rightsquigarrow_r$ and Lemma 4, if $\texttt{X} \rightsquigarrow_r \texttt{Y}$ and $r \approx r'$ then $\texttt{X} \rightsquigarrow_{r'} \texttt{Y}$.

We are now ready to use Theorem 6 to show that if a run does not contain a message chain from one operation to another operation, then operations in the run can be reordered so that the former operation takes place strictly after the latter one. More formally:

▶ **Theorem 11** (Moving one operation ahead of the other). *Let* $\texttt{X}$ *and* $\texttt{Y}$ *be two operations in a run* r*. If* $\texttt{Y}$ *completes in* r *and* $\texttt{X} \not\rightsquigarrow_r \texttt{Y}$*, then there exists a run* $r' \approx r$ *in which both (i)* $\texttt{Y} <_{r'} \texttt{X}$ *and (ii)* $\texttt{X} <_{r'} \texttt{Z}$ *holds for every completing operation* $\texttt{Z}$ *in* r *such that* $\texttt{X} <_r \texttt{Z}$ *and* $\texttt{Z} \not\rightsquigarrow_r \texttt{Y}$*.*

Proof. Let r' be the run built in the proof of Theorem 6 wrt. the run r with pivot $\theta = \texttt{Y}.e$ and delay $\Delta = t_{\texttt{Y}.e}(r) - t_{\texttt{X}.s}(r) + 1$. By Theorem 6 we have that $r \approx r'$, so each process performs the same operations and in the same local order. By the assumption, $\texttt{X} \not\rightsquigarrow_r \texttt{Y}$, i.e., $\texttt{X}.s \not\rightsquigarrow_r \texttt{Y}.e$, so $\texttt{X}$ is moved forward by Δ while $\texttt{Y}$ happens at the same real time in both r and r'. We thus have that $\texttt{Y} <_{r'} \texttt{X}$ because

$$t_{\texttt{X}.s}(r') = t_{\texttt{X}.s}(r) + \Delta = t_{\texttt{X}.s}(r) + t_{\texttt{Y}.e}(r) - t_{\texttt{X}.s}(r) + 1 = t_{\texttt{Y}.e}(r) + 1 = t_{\texttt{Y}.e}(r') + 1 > t_{\texttt{Y}.e}(r').$$

[3] We assume for simplicity that following an $\texttt{invoke}_i$, the environment will not issue another $\texttt{invoke}_i$ to the same process before i has provided a matching response.

Finally, let $\mathtt{Z}$ be an operation in r such that $\mathtt{Z} \not\rightsquigarrow_r \mathtt{Y}$ and $\mathtt{X} <_r \mathtt{Z}$. Since $\mathtt{Z} \not\rightsquigarrow_r \mathtt{Y}$, the real times of both $\mathtt{X}.e$ and $\mathtt{Z}.s$ in r' are shifted by Δ relative to their times in r. Thus, $\mathtt{X} <_r \mathtt{Z}$ implies that $\mathtt{X}$ ends before $\mathtt{Z}$ starts in r' also, i.e., $\mathtt{X} <_{r'} \mathtt{Z}$. ◀

6 Registers and Linearizability

A *register* is a shared object that supports two types of operations: *reads* R and *writes* W. We focus on implementing a MWMR (multi-writer multi-reader) register, in which every process can perform reads and writes, in an asynchronous message-passing system. Simulating a register in an asynchronous system has a long tradition in distributed computing, starting with the work of [4]. When implementing registers in the message passing model, one typically aims to mimic the behaviour of an atomic register. A register is called *atomic* if its read and write operations are instantaneous, and each read operation returns the value written by the most recent write operation (or some default initial value if no such write exists). The standard correctness property required of such a simulation is Herlihy and Wing's *linearizability* condition [15]. Roughly speaking, an object implementation is linearizable if, although operations can be concurrent, operations behave as if they occur in a sequential order that is consistent with the real-time order in which operations actually occur: if an operation $\mathtt{O}$ terminates before an operation $\mathtt{O}'$ starts, then $\mathtt{O}$ is ordered before $\mathtt{O}'$. More formally:

We denote by $\mathtt{invoke}_i(W, v)$ the invocation of a write operation of value v at process i and by $\mathtt{return}_i(W)$ the response to a write operation. (Recall that the invocation is an external input that process i receives from the environment, while the response is an action that i performs.) Similarly, $\mathtt{invoke}_i(R)$ denotes the invocation of a read operation at process i and by $\mathtt{return}_i(R, v)$ the response to a read operation returning value v. We say that an invocation $\mathtt{invoke}_i(\cdot)$ and a response $\mathtt{return}_i(\cdot)$ are *matching* if they both are by the same process and in addition, they both are invocation and response of an operation of the same type.

▶ **Definition 12** (Sequential History). *A sequential history is a sequence $H = S_0, S_1, \ldots$ of invocations and responses in which the even numbered elements S_{2k} are invocations and the odd numbered ones are responses, and where S_{2k} and S_{2k+1} are matching invocations and responses whenever S_{2k+1} is an element of H.*

We use the following notation:

▶ **Notation 1.** *Let H be a sequential history and let $\mathtt{X}, \mathtt{Y}$ be two operations in H. We denote $\mathtt{X} <_H \mathtt{Y}$ the fact that $\mathtt{X}$'s response appears before $\mathtt{Y}$'s invocation in H.*

▶ **Definition 13.** *An* atomic register history *is a sequential history H in which every read operation returns the most recently written value, and if no value is written before the read, then it returns the default value $\perp$.*

▶ **Definition 14** (Linearization). *A* linearization of a run r *is an atomic register history H satisfying the following.*

- *The elements of H consist of the invocations and responses of all completed operations in r, possibly some invocations of pending operations in r, and for each invocation of a pending operation that appears in H, a matching response.*
- *If $\mathtt{X} <_r \mathtt{Y}$ and the invocation of $\mathtt{Y}$ appears in H, then $\mathtt{X} <_H \mathtt{Y}$.*

▶ **Definition 15** (Linearizable Protocols). *P is a (live) linearizable atomic register protocol (*l.a.r.p.*) if for every run r of P:*

- *every operation invoked at a nonfaulty process in r completes, and*
- *there exists a linearization of r as defined above.*

Unless explicitly mentioned otherwise, all of the runs r in our formal statements below are assumed to be runs of an l.a.r.p. P.

7 Communication Requirements for Linearizable Registers

In this section, we study the properties of linearizable atomic register protocols in the asynchronous message passing model. Since linearizability is local [15], it suffices to focus on implementing a single register, since a correct implementation will be compatible with linearizable implementations of other registers and objects. We assume for ease of exposition that a given value can be written to the register at most once in any given run. (It follows that if the value v is written in r, we can denote the write operation by $\mathtt{W}(v)$).

We say that an operation $\mathtt{X}$ is a *v-operation* and write $\mathtt{X}v$ if (i) $\mathtt{X}$ is a read that returns value v, or (ii) $\mathtt{X}$ is a write operation writing v. In every linearization history of a run r of an l.a.r.p., a read operation returning a value $v \neq \bot$ must be preceded be an operation writing the value v. A direct application of Theorem 11 allows us to formally prove that, as expected, a read operation returning v must receive a message chain from the operation writing v:

▶ **Lemma 16.** *If a read operation $\mathtt{X}v$ in r returns a value $v \neq \bot$ then $\mathtt{W}(v) \rightsquigarrow_r \mathtt{X}v$.*

Proof. Let r be a run of a l.a.r.p. P, and assume by way of contradiction that there is an operation $\mathtt{X}v$ with $v \neq \bot$ in r such that $\mathtt{W}(v) \not\rightsquigarrow_r \mathtt{X}v$. Since $\mathtt{X}v$ is assumed to return v, it completes in r. Applying Theorem 11 wrt. $\mathtt{X} = \mathtt{X}v$ and $\mathtt{Y} = \mathtt{W}(v)$ we obtain a run $r' \approx r$ such that $\mathtt{X}v <_{r'} \mathtt{W}(v)$. By Lemma 4(ii) we have r' is a run of P as well. It follows that r' must have a linearization H. But by linearizability, H must be such that $\mathtt{X}v <_H \mathtt{W}(v)$. Since v is written only once, there is no write of v before $\mathtt{X}v$ in H, contradicting the required properties of a linearization. ◀

Lemma 16 proves an obvious connection: For a value to be read, someone must write this value, and the reader must receive information that this has occurred. But as we shall see, linearizability also forces the existence of other message chains; indeed, most pairs of operations in an execution must be related by a message chain.

A straightforward standard but very useful implication of linearizability for atomic registers is captured by the following lemma.

▶ **Lemma 17** (no a-b-a). *Let $\mathtt{X}a <_r \mathtt{Y}b <_r \mathtt{Z}c$ be three completing operations in a run r of a* l.a.r.p. *P. If $a \neq b$ then $a \neq c$.*

Proof. We first show the following claim:

▷ Claim 18. Let $\mathtt{R}v$ be a completing read operation occurring in r and let H be a linearization of r. Then (i) $\mathtt{W}(v) <_H \mathtt{R}v$, and moreover (ii) there is no value $v' \neq v$ s.t. $\mathtt{W}(v) <_H \mathtt{W}(v') <_H \mathtt{R}v$.

Proof. Recall that the sequential specification of a register states that a read must return the most recent written value. The fact that the value v must have been written implies (i). The fact that it is the last written value linearized before $\mathtt{R}v$ implies (ii). ◁

Returning to the proof of Lemma 17, let H be a linearization of r. Clearly, the real time order requirement of linearizability implies that $\mathtt{X}a <_H \mathtt{Y}b <_H \mathtt{Z}c$. By Claim 18 (i), we have that $\mathtt{W}(a) \leq_H \mathtt{X}a$ and $\mathtt{W}(b) \leq_H \mathtt{Y}b$. Combining these inequalities with Claim 18 (ii), we obtain that $\mathtt{W}(a) \leq_H \mathtt{X}a <_H \mathtt{W}(b) \leq_H \mathtt{Y}b \leq_H \mathtt{Z}c$. If $\mathtt{Z}c$ is a write operation then $a \neq c$ results from the fact that $\mathtt{W}(a) <_H \mathtt{Z}c$ and that the value a can be written at most once in r. If $\mathtt{Z}c$ is a read operation, then it cannot return a since the value a is not the last written value before $\mathtt{Z}c$ (since $\mathtt{W}(a) <_H \mathtt{W}(b)$). ◀

Lemmas 16 and 17 explain the second communication round of the ABD algorithm [4], also known as *Write-Back*: Roughly speaking, the Write-Back of a read $\mathtt{R}$ returning value v guarantees that the reader *knows* that for every future read $\mathtt{R}'$, the run will contain a message chain from $\mathtt{W}(v)$ through $\mathtt{R}$ to $\mathtt{R}'$.

Based on Theorem 11 and Lemma 17, we are now in a position to prove our most powerful result about linearizable implementations of atomic registers, which shows that they must create message chains between operations of all types: Reads to writes, writes to writes, reads to reads and writes to reads. Intuitively, Theorem 19 shows that if a value b is read, then every b-operation must be reached by a message chain from all other earlier operations.

▶ **Theorem 19** (Linearizability entails message chains). *Let $\mathtt{R}b$ be a completing read operation in r and let $\mathtt{Y}b$ be a b-operation that completes in r such that $\mathtt{R}b \not\rightsquigarrow \mathtt{Y}b$. Then for every $c \neq b$ and operation $\mathtt{X}c <_r \mathtt{R}b$, the run r contains a message chain $\mathtt{X}c \rightsquigarrow_r \mathtt{Y}b$.*

Proof. Assume by way of contradiction that there is an operation $\mathtt{X}c <_r \mathtt{R}b$ such that $\mathtt{X}c \not\rightsquigarrow_r \mathtt{Y}b$. First notice that all three operations $\mathtt{X}c$, $\mathtt{Y}b$ and $\mathtt{R}b$ complete in r', since $\mathtt{R}b$ and $\mathtt{Y}b$ complete by assumption and $\mathtt{X}c <_r \mathtt{R}b$. We apply Theorem 11 wrt. $\mathtt{X} = \mathtt{X}c$ and $\mathtt{Y} = \mathtt{Y}b$ and obtain a run $r' \approx r$ such that $\mathtt{Y}b <_{r'} \mathtt{X}c$. Moreover, since $\mathtt{X}c <_r \mathtt{R}b$, we also have by Theorem 11 (ii) that $\mathtt{X}c <_{r'} \mathtt{R}b$. We thus obtain $\mathtt{Y}b <_{r'} \mathtt{X}c <_{r'} \mathtt{R}b$ for values $b \neq c$. This contradicts Lemma 17, completing the proof. ◀

Intuitively, Theorem 19 shows that read or write operations involving a value that is actually read (i.e., returned by a read operation) must receive message chains from practically all earlier operations. We can show that the same can be true more broadly, e.g., even for a completing write operation $\mathtt{W}(v)$ where v is never read in the run.

▶ **Corollary 20.** *Let $\mathtt{X}a <_r \mathtt{Y}b$ and assume that $\mathtt{Y}b$ completes in r. If $\mathtt{Y}b$ runs in isolation in r and $a \neq b$, then $\mathtt{X}a \rightsquigarrow_r \mathtt{Y}b$.*

Proof. Let r be a run satisfying the assumptions. There exists a run r' such that (i) r' is identical to r up to $t_{Yb.e}(r)$ (in particular, $r'(m) = r(m)$ for all $0 \leq m \leq t_{\mathtt{Y}b.e}$), and (ii) there is an invocation of a read operation $\mathtt{R}$ in round $t_{\mathtt{Y}b.e} + 1$ of r', at a process i that is nonfaulty in r'. Since i is nonfaulty, $\mathtt{R}$ completes in r'. Moreover, since $\mathtt{Y}b$ runs in isolation and $\mathtt{R}$ starts after $\mathtt{Y}b$ ends, the value returned by $\mathtt{R}$ must be b. We obtain a run r' in which $\mathtt{Y}b <_{r'} \mathtt{R}b$ and $\mathtt{X}a <_{r'} \mathtt{R}b$ with $a \neq b$. So by Theorem 19 we have that $\mathtt{X}a \rightsquigarrow_{r'} \mathtt{Y}b$. Since $r'(m) = r(m)$ for all $0 \leq m \leq t_{\mathtt{Y}b.e}$ it follows that $\mathtt{X}a \rightsquigarrow_r \mathtt{Y}b$, as claimed. ◀

8 Failures and Quorums

By assumption, invocations of reads and writes to a register are spontaneous events, which is modeled by assuming that they are determined by the adversary (or the environment in our terminology) in a nondeterministic fashion. Intuitively, in a completing register implementation, the adversary can at any point wait for all operations to return and then perform a read. Suppose that this read operation is invoked at time t and that the value it returns is v. Then, by Theorem 19, the resulting run r must contain message chains $\mathtt{X} \rightsquigarrow_r \mathtt{W}(v)$ from *every* operation X that completed before time t to the write operation $\mathtt{W}(v)$. Therefore, before it can complete, every operation X must ensure that message chains from X to future operations can be constructed. There are several ways to ensure this in a reliable system. One way is by requiring the process on which X is invoked to construct a message chain to all other processes before X returns. This essentially requires a broadcast to all processes that starts after X is invoked. Another way to ensure this is by having every transaction Y coordinate a convergecast to it from all processes, that is initiated after Y is invoked. Each of these can be rather costly. A third, and possibly more cost effective way can be to assign a distinguished coordinator process c for the register object, and ensure that every operation X creates a message chain to c that is followed by a message chain back from c to the process invoking X. Notice that none of these strategies can be used in a system in which one or more processes can crash: After a crash, neither the broadcast nor the convergecast would be able to complete. Similarly, a coordinator c as described above would be a single point of failure, and once it crashes no operation could complete.

We now show that in a system in which up to f processes can crash, Theorem 19 implies that an operation must complete round-trip communications with at least f other processes before it can terminate. We proceed as follows.

▶ **Definition 21.** *We say that a process p* observes *a completed operation* X *in a run r if r contains a message chain from* $\mathtt{X}.s$ *to* $\langle p, t_{\mathtt{X}.e}\rangle$*. (The message chain reaches p by the time operation* X *completes.) Process p is called a* witness for X *in r if r contains a message chain from* $\mathtt{X}.s$ *to* $\mathtt{X}.e$ *that contains a p-node $\theta = \langle p, t\rangle$.*

▶ **Lemma 22.** *Let P be an f-resilient* l.a.r.p.*, and let* X *be a completed operation in a run r of P. Then more than f processes must observe* X *in r.*

Proof. Assume, by way of contradiction, that no more than f processes observe X in r. Let r' be a run of P that coincides with r up to time $t_{\mathtt{X}.e}$, in which all processes that have observed X fail from round $t_{\mathtt{X}.e} + 1$ (and no other process crashes), in which all operations that are concurrent with X complete and, after they do, a write operation $\mathtt{W}(v)$ (for a value v not previously written) runs in isolation, followed by a completed read. Since all processes that observed X in r' crash before $\mathtt{W}(v)$ is invoked, $\mathtt{X} \not\rightsquigarrow_{r'} \mathtt{W}(v)$. The read returns v, and so Theorem 19 implies that $\mathtt{X} \rightsquigarrow_{r'} \mathtt{W}(v)$, contradiction. ◀

We can now show that in f-resilient l.a.r.p.'s, every operation must perform at least one round-trip communication to all members of a quorum set of size at least f. Formally:

▶ **Theorem 23.** *Let P be an f-resilient* l.a.r.p.*, and let* X *be a completed operation in a run r of P. Then r must contain more than f witnesses for* X*.*

Proof. Assume by way of contradiction that there is a run r of P that contains $\leq f$ witnesses for X. Notice that for every witness p for X in r there must be a node $\langle p, t\rangle \rightsquigarrow_r \mathtt{X}.e$. We apply Theorem 6 to r with pivot $\mathtt{X}.e$ and delay $\Delta = t_{\mathtt{X}.e} - t_{\mathtt{X}.s} + 1$, to obtain a run $r' \approx r$. By Lemma 4(iii) the run r' is a run of P. By choice of Δ, only processes with nodes in $\mathsf{past}_{r'}(\mathtt{X}.e)$ can observe X in r', so every observer of X must be a witness for X. By construction $\mathsf{past}_r(\mathtt{X}.e) = \mathsf{past}_{r'}(\mathtt{X}.e)$, and so there are no more than f witnesses for X in r'. It follows that no more than f processes observe X in r', contradicting Lemma 22 . ◀

The ABD algorithm requires the number of processes to satisfy $n \geq 2f + 1$ [4]. This ensures that every two sets of $n - f$ processes intersect in at least one process, i.e., each operation communicates with a quorum set. We remark that although Theorem 23 implies the need to communicate with quorum sets, the Write-Back round is not *always* necessary. If a reader of v receives message chains from all processes that are in a quorum set that $\mathtt{W}(v)$ communicated with in the first round, then the message chains of Lemma 16 can be guaranteed without the Write-Back. The algorithm of [10] is based on this type of observation. In addition, strengthening the results of [21], our work implies that the channels that are shown to exist in [21] must in fact be used to interact with quorums.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993. doi:10.1145/153724.153741.

2 Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 51–60, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3212734.3212741.

3 Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241*, 2003.

4 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995. doi:10.1145/200836.200869.

5 Naama Ben-David, Michal Friedman, and Yuanhao Wei. Survey of persistent memory correctness conditions. *arXiv preprint arXiv:2208.11114*, 2022. doi:10.48550/arXiv.2208.11114.

6 François Bonnet and Michel Raynal. A simple proof of the necessity of the failure detector sigma to implement an atomic register in asynchronous message-passing systems. *Information Processing Letters*, 110(4):153–157, 2010. doi:10.1016/j.ipl.2009.11.011.

7 K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986. doi:10.1007/BF01843569.

8 Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 338–346, 2004. doi:10.1145/1011767.1011818.

9 Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Andreas Tielmann. The weakest failure detector for message passing set-agreement. In *International Symposium on Distributed Computing*, pages 109–120. Springer, 2008. doi:10.1007/978-3-540-87779-0_8.

10 Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 236–245, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1011767.1011802.

11 Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3342195.3387543.

12 Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about Knowledge.* MIT Press, Cambridge, Mass., 2003.

13 Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 373–382, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993636.1993687.

14 Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming.* Newnes, 2020.

15 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

16 Kaile Huang, Yu Huang, and Hengfeng Wei. Fine-grained analysis on fast implementations of distributed multi-writer atomic registers. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 200–209, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405698.

17 Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *ACM SIGPLAN Notices*, 47(8):141–150, 2012. doi:10.1145/2145816.2145835.

18 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. doi:10.1145/359545.359563.

19 N.A. Lynch and A.A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 272–281, 1997. doi:10.1109/FTCS.1997.614100.

20 Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996. doi:10.1145/248052.248106.

21 Alejandro Naser-Pastoriza, Gregory Chockler, and Alexey Gotsman. Fault-Tolerant Computing with Unreliable Channels. In Alysson Bessani, Xavier Défago, Junya Nakamura, Koichi Wada, and Yukiko Yamauchi, editors, *27th International Conference on Principles of Distributed Systems (OPODIS 2023)*, volume 286 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:21, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.OPODIS.2023.21.

22 Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. SwiftPaxos: Fast Geo-Replicated state machines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 345–369, Santa Clara, CA, April 2024. USENIX Association. URL: https://www.usenix.org/conference/nsdi24/presentation/ryabinin.

23 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. doi:10.1145/98163.98167.

A Detailed Model

Our model is based on the asynchronous message passing model of Fagin et al. [9]. We consider a set Π of n processes. connected via a communication network Net, defined by a directed graph (Π, E). Every edge $(i, j) \in E$ is associated with a *channel*, and denoted by $\mathtt{chan}_{ij}$, consisting of a set of records of the form $|\mu, t|$. Such a record represents the fact that μ was sent by i to j in round t and is still in transit (i.e., it has not been delivered yet). In addition to the processes, the *environment* (denoted by e) models what is commonly referred to as the adversary.

- **Actions:** For $i \in \Pi$, the set of actions Act_i it can perform consists of local actions α_i (possibly including a no_op_i action) and message send actions of the form $send_i(\mu, j)$. The environment actions will play the role of determining when messages are delivered, when processes are scheduled to move and external inputs to individual processes. Thus, an environment action α_e is a tuple $\vec{\eta} = \langle \eta_1, ..., \eta_n \rangle$ containing a component η_i for every process i. Each η_i is either $\mathtt{move}_i$, $\mathtt{skip}_i$, $\mathtt{invoke}_i(x)$, or $\mathtt{deliver}_i(|\mu, t|, j)$ for some message μ and process $j \neq i$. The $\mathtt{move}_i$ action means that i performs an action according to its protocol as defined below; $\mathtt{skip}_i$ means that i is ignored in the current round; $\mathtt{invoke}_i(x)$ means that i will receive the external input x, while $\mathtt{deliver}_i(|\mu, t|, j)$ means that i will receive the message μ from j, provided that this message is in transit, and is the next message to be delivered in FIFO order.

- **States:** The local state of a process at any given point is its local event history h_i, containing an initial value and a sequence of local events: all the messages i received, external inputs and all the actions i performed, all arranged in the order in which i observed them. The local state of the environment contains the complete record of all actions performed so far, as well as the current contents of $\mathtt{chan}_{ij}$ for all network edges $(i, j) \in E$. A global state is a tuple $g = (\ell_e, \ell_1, \dots, \ell_n)$ containing a state for the environment and a state for each one of the processes. An *initial* global state is one in which all local states contain only the initial values.
- **Protocols:** A *protocol* P_i associates a nonempty set of actions (of Act_i) with every local state of the process i. If $P_i(\ell_i) = S$, then the action performed by i when scheduled to move in state ℓ_i will be one of the elements of S. A protocol for the processes has the form $P = (P_1, \dots, P_n)$, and it associates a protocol P_i with every process $i \in \Pi$.
- **Environment Protocol:** The environment's protocol, which we denote by P_e^a, is given by $P_e^a(\ell_e) \triangleq \{\vec{\eta} : \vec{\eta} \in Act_e\}$. In words, $P_e^a(\ell_e)$ performs, for every process $i \in P$, an independent, nondeterministic choice of η_i among the possibilities of $\mathtt{move}_i$, $\mathtt{skip}_i$, $\mathtt{deliver}_i(\cdot, \cdot)$ or $\mathtt{invoke}_i(\cdot)$.
- **Transition Function:** A joint action is a tuple $(\vec{\eta}, \alpha_1, \dots, \alpha_n)$ with $\vec{\eta} \in Act_e$ and $\alpha_i \in Act_i$ for each $i \in \Pi$. The transition function modifies the environment's local state ℓ_e by appending the current round's joint action at the end of the event history h. Local states are transformed as follows: If $\eta_i = \mathtt{move}_i$ then the action $\alpha_i \in P_i(\ell_i)$ that i performs (as recorded in the joint action added to h) is appended at the end of h_i. Moreover, if $\eta_i = \mathtt{move}_i$ and i's action is $send_i(\mu, j)$ where (i, j) is a link in Net, then a record $|\mu, m|$, where m is the current (sending) round, is added to $\mathtt{chan}_{ij}$. Similarly, if $\eta_i = \mathtt{invoke}_i(\cdot)$, then this external input is appended at the end of h_i. If $\eta_i = \mathtt{deliver}_i(|\mu, t|, j)$ and $|\mu, t|$ is the oldest message in $\mathtt{chan}_{ji}(m)$ (the message μ was sent in round t by j, is still in transit at the current time m, and is the next message to be delivered according to FIFO order), then this record $|\mu, t|$ is removed from $\mathtt{chan}_{ji}$ and (j, μ) is appended to the end of h_i. In this case, μ is said to be delivered in round m in r. The local state ℓ_i of $i \in P$ remains unchanged if $\eta_i = \mathtt{skip}_i$ or if $\eta_i = \mathtt{deliver}_i(|\mu, t|, j)$ and $|\mu, t| \notin \mathtt{chan}_{ji}(m)$, and the i-component in the joint action $\vec{\alpha}_m$ performed at time m is '$\bot_i$'.
- **Runs:** A run r is an infinite sequence of global states, whose first element $r(0)$ is the initial global state and we use $r(m)$ to denote the $(m + 1)$th state in the sequence. We identify time with the natural numbers, and think of $r(m)$ as being the global state at time m in r. We denote by $r_i(m)$ the local state of process i in $r(m)$.
- **Run of a protocol $\boldsymbol{P}$:** A run r is called a run of P if
  - **(i)** $r(0)$ is an initial global state, and
  - **(ii)** for every $m \geq 0$, there is a joint action $\vec{\alpha} = (\vec{\eta}, \alpha_1, \dots, \alpha_n)$ with $\vec{\eta} \in Act_e$ and $\alpha_i \in P_i(r_i(m))$ for every $i = 1, \dots, n$ such that $r(m + 1)$ is obtained by applying the transition function to $\vec{\alpha}$ and $r(m)$.

A protocol P_i is deterministic if it always specifies a unique action, i.e., if $P_i(\ell_i) = S$ then $|S| = 1$. We remark that while processes may or may not follow a deterministic protocol, the environment's protocol is highly nondeterministic. The asynchronous aspect of an a.m.p. is captured mainly by the environment's protocol and the transition function: The environment can delay the delivery of a message for arbitrarily long, and it can similarly delay a process from taking a step, and this is independent of how many steps others take, and of whether they receive messages sent to them. Moreover, the transition function is such that a process' local state changes only if the process either receives a message, takes a step, or that the environment's action is an invocation. Thus, it has no way of telling whether and how much time has passed since its last move.

Crashes and loss of messages. A process i is said to be correct in r if it is allowed to move ($\eta_i = \texttt{move}_i$) infinitely often in r. Otherwise process i is *faulty* (or *crashes*) in r. We say that a message μ is *lost* in r if it is sent in r and never delivered.

A system is said to be *reliable* if no process ever fails and no message is ever lost, in any of its runs. A protocol is said to be f*-resilient* if it acts correctly in all runs in which no more than f processes are faulty.

We now restate and prove Lemma 4.

▶ **Lemma 4.** *Let r and r' be two runs such that $r \approx r'$. Then*

(i) *If $\theta_1 \rightsquigarrow_r \theta_2$ then $\theta'_1 \rightsquigarrow_{r'} \theta'_2$ holds for all nodes θ'_1 and θ'_2 of r' such that $\theta_1 \sim \theta'_1$ and $\theta_2 \sim \theta'_2$*

(ii) *If r is a run of protocol P, then r' is also a run of P*

(iii) *A process i fails in r iff it fails in r', and similarly*

(iv) *A message μ is lost in r iff the same message is lost in r'*

Proof. We prove each one of the claims.

(i) Let θ_1, θ_2 in r such that $\theta_1 \rightsquigarrow_r \theta_2$. Denote by $\theta_1 = \alpha_1, \alpha_2, \ldots, \alpha_k = \theta_2$ the nodes constituting this message chain such that α_{i+1} is obtained from α_i applying (1a) or (1b) of Definition 1. We prove by induction on k that α_k has a corresponding message chain $\alpha'_1 \rightsquigarrow_{r'} \alpha'_k$ in r' when $\alpha_1 \sim \alpha'_1$ and $\alpha_k \sim \alpha'_k$.
Base: $k = 1$. The base case results directly from the fact that every local state in r appears in r' and vice-versa. Thus there is $\alpha'_1 \sim \alpha_1$ in r'.
Step: Let $k > 1$ and assume inductively that the claim holds for $\theta_1 = \alpha_1, \alpha_2, \ldots, \alpha_{k-1}$. If α_k is obtained from α_{k-1} by (1a) of Definition 1, then let $\alpha'_{k-1} = \langle p, t_{k-1}\rangle$ be the node of r' such that $\alpha_{k-1} \sim \alpha'_{k-1}$. By local equivalence between r and r' there must be a node $\alpha'_k \sim \alpha_k$ of p in r', and the claim holds.
If α_k is obtained from α_{k-1} by (1b), meaning that a message is sent at node α_{k-1} and arrives no later than at $\alpha_k = \langle p, t_k\rangle$, then since the send and the delivery of messages are registered in processes local states, we have by definition of locally equivalence that this message is also sent at α'_{k-1} and arrives no later than a node $\alpha'_k \sim \alpha_k$ of p in r'.

(ii) Assume r is a run of P. We show that if action α_i is performed in r' then it is an action of P. Let α'_i be an action performed by i in r' and denote by l_i the state of i right after performing this operation . By definition of local equivalence, there is a point in r such that i has local state l_i in r. I.e., i performed action α'_i in r, which by assumption is a run of P.

(iii) If a process i fails in r, then there is a time t from which the environment action of i is not $\texttt{move}_i$ anymore. Thus, there is a finite number of actions registered to its local state along the run. By definition of local equivalence, this is the case also for r', i.e., i fails also in r'.

(iv) Let μ be a message sent by i to j at node θ in r. It follows from item (i) that μ is sent at some node $\theta' \sim \theta$ in r'. If μ is lost in r, then this message is never delivered to process j and thus this reception is never added to the local state of j. By item (i) it follows μ is also lost in r'. ◀

Single Bridge Formation in Self-Organizing Particle Systems

Shunhao Oh[1]
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

Joseph L. Briones
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Jacob Calvert
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

Noah Egan
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

Dana Randall
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

Andréa W. Richa
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Abstract

Local interactions of uncoordinated individuals produce the collective behaviors of many biological systems, inspiring much of the current research in programmable matter. A striking example is the spontaneous assembly of fire ants into "bridges" comprising their own bodies to traverse obstacles and reach sources of food. Experiments and simulations suggest that, remarkably, these ants always form one bridge – instead of multiple, competing bridges – despite a lack of central coordination. We argue that the reliable formation of a single bridge does not require sophistication on behalf of the individuals by provably reproducing this behavior in a self-organizing particle system. We show that the formation of a single bridge by the particles is a statistical inevitability of their preferences to move in a particular direction, such as toward a food source, and their preference for more neighbors. Two parameters, η and β, reflect the strengths of these preferences and determine the Gibbs stationary measure of the corresponding particle system's Markov chain dynamics. We show that a single bridge almost certainly forms when η and β are sufficiently large. Our proof introduces an auxiliary Markov chain, called an "occupancy chain," that captures only the significant, global changes to the system. Through the occupancy chain, we abstract away information about the motion of individual particles, but we gain a more direct means of analyzing their collective behavior. Such abstractions provide a promising new direction for understanding many other systems of programmable matter.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Self-organizing particle systems, programmable matter, bridging, jump chain

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.34

Related Version *arXiv Version*: `https://arxiv.org/abs/2408.10830` [17]

1 Introduction

Ants are known to collaborate to form elaborate structures. Army ants of genus *Eiticon* self-assemble and disperse while foraging to form shortcuts in order to more efficiently reach food [19]. Weaver ants of species *Oecophylla longinoda* bind leaves together when building

[1] Corresponding author.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 34; pp. 34:1–34:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

their nests and can form long chains [13]. In this work, we take inspiration from the behavior of fire ants of species *Solenopsis invicta*, which can self-assemble into floating rafts and structural bridges [16]. These bridges are structures composed of the ants as they entangle themselves and allow other ants a medium over which they can reach a target, such as food. It is not well understood why individual ants sacrifice themselves by becoming entangled with others in unfavorable circumstances such as suspension over water, but such occurrences are regularly accomplished without any centralized coordination.

Recent laboratory experiments at the University of Georgia [24] were the first to replicate the more general fire ant bridging found in nature by placing food in the center of a bowl filled with water to see how hungry ants will coordinate. Initially, ants surround the bowl looking for food, but soon they begin to explore the water, begin building multiple structures extending from the edge of the bowl. Eventually some structures start to reach the food, but over time only a *single bridge* persists, with the other ants traversing the bridge to bring food back to the nest. Biologists have questioned what coordination allows the ants to always form a single bridge, streamlining their initial structures to minimize the number of ants sacrificed for the bridge while enabling the remaining ants to productively harvest food for the nest [19]. Physicists, biologists and computer scientists [4] have developed an agent-based model that mimics this behavior with locally interacting particles that primarily depends on two parameters: one capturing their individual awareness of the food source, based on proximity, and another that captures the rigidity of the bridge structure by measuring an interparticle attraction among all nearest neighbor pairs, which has been shown to suffice to encourage collectives to aggregate [4]. This physical model also allows ants to climb over one another and incorporates additional physical parameters such as the meniscus effects at the edge of the bowl and effects of small clusters disconnecting. Their simulations reliably replicate the tendency for exactly one bridge to form, but lack an explanation because the biological and simulated protocols are too difficult to analyze in order to connect the local behavior of individuals with the complex coordinated structure formed by the ensemble.

We show that the emergence of at most one bridge, as observed in ant experiments, follows from a statistical inevitability, and not from the sophistication of the individuals. Here, rather than tracking the movements of individual particles (or ants) over time according to a local Markov chain, we define a related auxiliary *occupancy chain* that captures only the changes to which sites are occupied. The states of the occupancy chain record the sites where one or more particles currently exist and the transitions capture the stochastic changes to the occupied sites over time, to determine long-term behavior. This approach of using occupancy chains allow us to apply techniques from statistical physics to infer properties of contours, and the simplified analysis is likely to be useful in other contexts, such as various fixed magnetization spin systems [8], biased growth processes [9], and collectives arising in swarm robotics responding to directed external stimuli [12].

1.1 Related work

Ant bridging can be viewed as an example of programmable matter, in which researchers seek to create materials that can be programmed to change their physical properties in response to user input and environmental stimuli. Diverse examples of programmable matter – like DNA tiles [22], synthetic cells [11], and reconfigurable modular robots [5] – are linked through their common use of local interactions to produce global functionality beyond the capabilities of individuals. Examples of this phenomenon abound in nature, including honey bees that select nest sites using decentralized recruitment [2], cockroach larvae that aggregate using short-range pheromones [10], and fire ants that assemble their bodies into rafts to

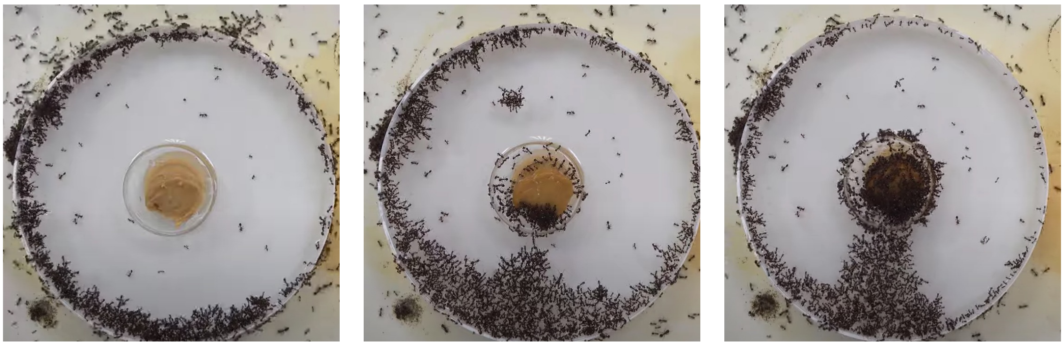


Figure 1 Photos of fire ants, *Solenopsis*, self-assembling over time to form a bridge to reach food placed in the center of a bowl filled with water. Experiments performed by and photo credit to Takao Sasaki and Horace Zeng at the University of Georgia.

survive floods [16]. For this reason, programmable matter is often designed by analogy with biological collectives that exhibit an emergent behavior of interest [20, 18]. However, this form of biomimicry is generally unamenable to rigorously proving that an algorithm executed by the constituents of matter reliably leads to a desired behavior. This analysis is critical to understanding the extent to which algorithms for programmable matter are robust when they are actually deployed [12].

An abstraction of programmable matter known as a *self-organizing particle system* (SOPS) addresses this gap. In the standard amoebot model for SOPS [7, 6], particles exist on the sites (or nodes) of a lattice, with at most one particle per site, and move between sites along lattice edges. Each particle is anonymous (unlabeled), interacts only with particles on adjacent lattice sites, has limited (normally assumed to be constant-size) memory, and does not have access to any global information such as a common direction on the lattice, a coordinate system or the total number of particles. We note that in this paper, we are studying a particle system with possibly multiple particles (ants) per site, but we reduce to an occupancy chain which recovers the "SOPS perspective" by just indicating whether each site is occupied or not. Recent work on SOPS has led to rigorous stochastic distributed algorithms for various tasks by exploring phase changes that lead to desirable collective behavior with high probability. Examples include compression and expansion [4], where particles cluster closely or expand; separation [3], where colored particles would like to separate into clusters of same color particles; aggregation [12], where the system would like to compress but no longer needs to remain connected; shortcut bridging [1]; locomotion [21, 23]; and transport [12]. However, while these algorithms based on local Markov chains provide insight into emergent collective behavior, the proofs guaranteeing the long-term behaviors become prohibitively challenging as the collectives take on increasingly complicated tasks with more sophisticated interactions. For example, while the work in [1] also looked at the bridging behavior of fire ants [19], it did not focus on the bridge formation process. Instead, the paper assumed that a single bridge already existed, anchored at two fixed points (corresponding to the nest and the food), and considered the (simpler) problem of optimizing the placement of the bridge with respect to parameters of relevance.

1.2 Overview of our results

To gain insight into structures that emerge from ant bridging, we consider a discrete model on a finite $w \times h$ domain $\Lambda = \Lambda(w, h)$ of the triangular lattice, with periodic boundary conditions in the horizontal direction, representing the circular bowl containing food. It

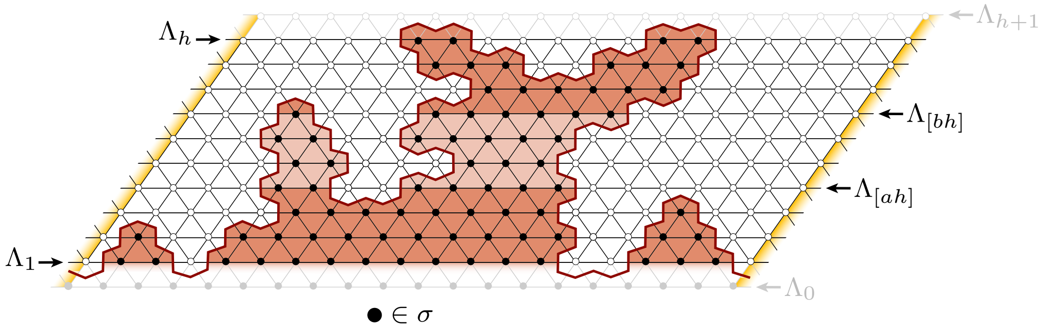


Figure 2 A domain Λ of the triangular lattice of width of $w = 20$ and height of $h = 10$ with periodic boundary conditions identifying the ends of rows (indicated by yellow shading). The configuration σ (black dots) has multiple (a, b)-bridges for $(a, b) = (0.4, 0.7)$, but not for $(a, b) = (0.4, 0.8)$.

will be convenient to think of Λ as a cylindrical domain comprised of h layers (or cycles), corresponding to the h rows Λ_1 through Λ_h, each of width w (see Figure 2). We model the motion of a finite set of n particles entering Λ at Λ_1, and growing a connected component by occupying sites on the interior of Λ in order to eventually reach the food, which we assume can be found at every site on Λ_h. Particles may move onto empty sites or walk upon sites currently occupied by other particles – we intentionally abstract away the specific dynamics by directly analyzing the occupancy chain, which only tracks changes to the set σ of occupied sites in Λ. In analogy with fire ant bridging, a step of the occupancy chain can be interpreted as a lone ant on the boundary of σ climbing over a neighboring ant, thus creating an unoccupied site, or as an ant concluding its walk over other ants to occupy a new site adjacent to the previous boundary. For simplicity, we assume that the aspect ratio $\alpha = w/h$ is a constant and that n scales linearly with the area of the region, writing $n = \lfloor \rho h^2 \rfloor$ for some constant $\rho > 0$, as these are the conditions under which bridges may form.

The occupancy chain captures the relative preferences of particles to move toward Λ_h (e.g., due to the scent of the food source placed along Λ_h), parameterized by $\eta >\ 0$, and their preference for locations with more neighbors, increasing the rigidity of the connected structure. The balance of these preferences is expressed through a function on configurations called the *Hamiltonian*, which is related to the stationary probability of configurations and expresses the overall "fitness" of a configuration, with each particle contributing its piece.

For a configuration σ and a site $v \in \sigma$, let $S(v)$ be the *intensity of the scent* at v, which is any nonincreasing function of graph distance $d(v)$ from v to Λ_h. Let $B_\sigma(v)$ be the *number unoccupied sites that neighbor* v. Then the Hamiltonian H of a configuration σ is

$$H(\sigma) = -\sum_{v\,\in\,\sigma} \left(\eta S(v) - B_\sigma(v)\right), \tag{1}$$

where $\eta > 0$ is a parameter describing how much a particle favors a stronger scent relative to how much it desires more neighbors. A second parameter β, the interparticle attraction, captures the strength of the Hamiltonian. These induce a probability distribution π over the set of valid configurations Ω (defined in Section 2.1) known as the Gibbs distribution, where

$$\pi(\sigma) = \exp\left(-\beta H(\sigma)\right)/Z, \quad \text{for } \sigma \in \Omega \tag{2}$$

in terms of the normalization constant $Z = \sum_{\tau\in\Omega} e^{-\beta H(\tau)}$. Note we have taken some liberties in defining the occupancy chain by assuming that each particle has been walking on the current bridge structure sufficiently long that it is near equilibrium.

We may now talk about the shape of the contour between the set of occupied and unoccupied sites on Λ in order to determine the number of bridges at equilibrium. We introduce a weak definition of "multiple bridges" to show that any contours that unnecessarily "backtrack" a nontrivial amount from *any* layer will be extremely unlikely when the interparticle attraction β is large enough. We say that a configuration σ has *multiple* (a, b)*-bridges* for real numbers $0 < a < b < 1$ if the restriction of σ to layers $\lfloor ah \rfloor$ and greater contains at least two components (on the lattice Λ) that reach layer $\lfloor bh \rfloor$. This is formally stated later in Definition 4 . Intuitively, one can imagine such configurations as ones that extend out to layer $\Lambda_{\lfloor bh \rfloor}$, retract back to $\Lambda_{\lfloor ah \rfloor}$, then extend out to $\Lambda_{\lfloor bh \rfloor}$ a second time.

Our first main theorem states that, on any $w \times h$ region Λ with constant aspect ration α, regardless of the scent parameter η, if the interparticle attraction β is large enough, then for any constant $\epsilon \in (0, 1)$, the *probability of seeing multiple* (a, b)*-bridges where* $b - a \geq \epsilon$ *is always exponentially small in* β *and* h (see Theorem 5 in Section 3.2). Taking $b = 1$ shows that we are unlikely to see a contour that reaches the food multiple times while backtracking a distance ϵ between the intervals in which they touch.

Our second main result concerns whether or not any bridge will be likely to form. This question is much more delicate and depends on the scent parameter η, some basic assumptions about the scent gradient (see Definition 9), and some bounds on the number of particles in the system. If the *scent parameter* η *is too small*, then particles will compress around the boundary and are unlikely to reach the food, so *no bridge will form.* If the *scent parameter is sufficiently large*, then a *single bridge will emerge, but never more.* These conditions are properly outlined in Section 3.2 and stated as Theorem 10 and Corollary 11.

The proofs are motivated by a technique from statistical physics known as a Peierls argument that constructs a mapping from low-probability configurations to high-probability ones, to infer global properties of the distribution π. The key challenge in constructing suitable mappings for our model is the dependence of the scent component S of the Hamiltonian on the distance of sites from Λ_h. As a consequence, standard physics approaches based on simple mappings that rearrange sites is insufficient. We overcome this challenge by introducing the *layer sequence* of a configuration, which counts the number of the configuration's elements in each layer. Its virtue is that it is far easier to describe and analyze transformations of layer sequences than those of configurations, and the layer sequence retains the information necessary to calculate the scent component of the configuration's Hamiltonian. Moreover, while it discards the geometric information that is needed to calculate the configuration's boundary length, we are nevertheless able to obtain useful bounds on the boundary length.

Our choice to model the motion of the contour directly via an occupancy chain, instead of inferring it from dynamics prescribed to individual particles (ants), is a significant and valuable departure from the norm in distributed computing. Without this abstraction tracking only the salient changes to the profile of the ensemble, it would be far more difficult to rigorously analyze the bridge formation behavior. Notably, this abstraction comes at the cost of obscuring the underlying particles' computational capabilities and the local distributed algorithm that govern the particles' dynamics, which we address and clarify in Section 2.2.

The success of SOPS in designing simple, local algorithms that produce complex, global behaviors suggests that the apparently intelligent behaviors of natural collectives may in fact be the inevitable consequences of unremarkable circumstances. We see a broader opportunity here, not only to explain how the intelligent behaviors of natural collectives arise, but also to demonstrate that such an explanation need not assume great intelligence or detailed knowledge on behalf of the individuals. We believe incorporating occupancy chains may be a way to provide insight into other complex collectives.

2 The Bridging Occupancy Chain

In the preceding section, we introduced the domain Λ, the Hamiltonian H, and the probability distribution π that H induces, to the extent necessary to informally state our results. In particular, we mentioned that π is supported on a subset of valid configurations Ω, but deferred a definition. We now introduce an extension of Λ that we use to define Ω, and then we elaborate the terms of the Hamiltonian, before defining π and the occupancy chain.

2.1 The model

Some subsets of the vertices of Λ are not suitable abstractions of the bridges in Figure 1, because they do not extend from Λ_1 or because they contain holes. To make this precise, we extend the domain Λ with two further layers, Λ_0 and Λ_{h+1}, the sites of which we respectively treat as always occupied and always unoccupied (Figure 2). We denote by $\overline{\Lambda}$ this extended domain, which is isomorphic to Λ with a height of $h+2$, as shown in Figure 2. We define the set of valid configurations to be

$$\Omega = \left\{\sigma \subseteq V(\Lambda) : \ |\sigma| \leq n \ \text{ and } \ \sigma \cup V(\Lambda_0) \text{ is simply connected in } \overline{\Lambda}\right\},$$

where $|\sigma|$ denotes the number of elements of σ.

Recall that the Hamiltonian (Equation 1) of a configuration $\sigma \subseteq V(\sigma)$ involves a sum over sites $v \in \sigma$, with contributions from the scent intensity $S(v)$ at v and the number $B_\sigma(v)$ of neighbors of v that are not in σ. To be precise, we define $B_\sigma(v)$ according to

$$B_\sigma(v) = |\{u \in V(\Lambda) \setminus \sigma : (u, v) \in E(\Lambda)\}|,$$

which serves to penalize configurations that are less compact. Note that, if $\sigma \in \Omega$, then $B_\sigma(v) < 6$ for every $v \in \sigma$. The scent intensity of a site $v \in V(\Lambda)$ is a function that depends only on its distance to the food, which we will assume is stronger the closer a site is to the food. For the sake of the proofs however, we will phrase these as distances $d(v, \Lambda_0)$ from the bottom of the region $\overline{\Lambda}$, noting trivially that $d(v, V(\Lambda_0)) = h - d(v, V(\Lambda_h))$. For convenience of notation, we define a nondecreasing function $S_h : [h] \mapsto \mathbb{R}_{\geq 0}$, and write the scent intensity at a vertex v as

$$S(v) = S_h\big(d(v, V(\Lambda_0))\big).$$

While the function S_h being nondecreasing is sufficient to show in Section 3.1 that multiple bridges are exponentially unlikely, when we discuss whether or not a bridge to the food will form in Section 3.2, we use a slightly narrower class of functions that we call *nondecelerating* (Definition 9). This covers a broad class of functions, including but not limited to:

$$S_h(y) = C_h y^k - D_h \quad \text{or} \quad S_h(y) = C'_h (h-y)^{-k'} - D'_h,$$

where $k, k' \geq 1$, where the values C_h, C'_h normalize the sum of S_h over a “column” of Λ to a constant, and D_h, D'_h are chosen to make $S_h(1) = 0$. The purpose of the normalization is to make the contributions of $S(v)$ and $B_\sigma(v)$ to the Hamiltonian comparable, so η controls whether a bridge forms. In the case of $k, k' = 1$, the first option gives us a linear scent gradient, while the second option represents a scent intensity proportional to the reciprocal of the distance from the food. Section 3.2 discusses the cases when this sum is not normalized.

We define the occupancy chain in the next section and show that it constitutes an aperiodic and irreducible Markov chain on Ω, hence it has a unique stationary distribution on Ω, as given in Equation 2. Observe that by writing $\lambda = e^{\beta} > 0$ and $\gamma = e^{\beta\eta} > 0$, this formulation can be equivalently written as

$$\pi(\sigma) \propto \prod_{v \in \sigma} \left(\lambda^{-B_{\sigma}(v)} \gamma^{S(v)} \right), \tag{3}$$

where the probability of each configuration can be written as a product of weights of its individual particles, normalized.

Although the Hamiltonian is partly inspired by the Ising model where the preference for having more neighbors can be thought of as the strength of a ferromagnetic interaction between particles, the scent component of the Hamiltonian introduces a directional bias that deviates from classical models, making the analysis considerably more challenging (see, e.g., [9]). Viewing the dynamics through the occupancy chain gives us a succinct tool to derive emergent behaviors of the collective not as easily provable for related agent-based models.

2.2 The algorithm

The bridging occupancy chain algorithm can be briefly described as follows: Initially, the sites of Λ are unoccupied. Given a configuration σ, we select a uniformly random site v of Λ. If v is occupied, then we attempt to remove it from σ. Otherwise, if v is unoccupied and $|\sigma|$ is less than n, then we attempt to add v to σ. The move is rejected if it produces a configuration that does not belong to Ω, i.e., if it is not simply connected in $\overline{\Lambda}$.

To enforce that simple connectivity is maintained at all times, starting from a simply connected configuration, we follow what we call *local simple connectivity* checks, that can be shown to ensure that the configuration as a whole always remains simply connected in $\overline{\Lambda}$ (Lemma 2). Note that the checks below only depend on a site v and its immediately adjacent sites in $\overline{\Lambda}$, which will be important when we discuss the underlying particle dynamics at the end of this section.

▶ **Definition 1** (Local Simple Connectivity). *For a site $v \in V(\Lambda)$, our definition of local simple connectivity depends on whether v is occupied or not. Let $\overline{\mathcal{N}}(\sigma, v)$ denote the extended neighborhood of v in $\sigma \cup V(\Lambda_0)$, where v is any site of Λ.*

1. *If $v \in \sigma$, we say that v is locally simply connected iff $|\overline{\mathcal{N}}(\sigma, v)| \leq 5$ and the induced subgraph $\overline{\Lambda}[\overline{\mathcal{N}}(\sigma, v)]$ is connected.*
2. *If $v \notin \sigma$, we say that v is locally simply connected iff $|\overline{\mathcal{N}}(\sigma, v)| \geq 1$ and the induced graph $\overline{\Lambda}[\overline{\mathcal{N}}(\sigma, v)]$ is connected.*

The intuition behind this definition is that in Case (1) it is safe to remove a locally simply connected site v from σ, since making v unoccupied cannot disconnect its neighborhood in σ (nor create a hole at v); in Case (2), making the site v occupied cannot connect previously disconnected neighboring sites in σ and therefore cannot create a hole. Note that, for every locally simply connected move, the reverse is also a valid move.

Algorithm 1 describes the moves of the occupancy chain. Note that we are using a variant of the Metropolis-Hastings algorithm [15], but it is easy to check that for any two configurations whose symmetric difference is a single vertex, detailed balance will be satisfied.

The following two lemmas prove the correctness of the Bridging Occupancy Chain. We start by proving that the algorithm will keep the configurations simply connected and then show the Bridging Occupancy Chain is irreducible and aperiodic, and therefore converges to the indicated stationary distribution π.

Algorithm 1 Bridging Occupancy Chain.

```
Let $\sigma$ denote the current configuration.
$v \leftarrow$ uniformly random site in $\Lambda$
$p \leftarrow$ uniformly random number in $[0, 1]$
if $v$ is locally simply connected then
    if $v \in \sigma$ and $p \leq \min\{1, \exp(\beta(2B_\sigma(v) - \eta S(v)))\}$ then
        Make $v$ unoccupied.
    else if $v \notin \sigma$, $|\sigma| < n$, and $p \leq \min\{1, \exp(-\beta(2B_\sigma(v) - \eta S(v)))\}$ then
        Make $v$ occupied.
```

▶ **Lemma 2** (Maintaining Simple Connectivity). *Adding or removing a locally simply connected site from a configuration $\sigma \in \Omega$ maintains simple connectivity.*

Proof. Let σ be the current configuration, which we define as simply connected if $\sigma \cup V(\Lambda_0)$ is simply connected in $\overline{\Lambda}$. As Λ_0 is treated as always filled, $\sigma \cup V(\Lambda_0)$ is never empty.

The only way the removal of an occupied site $v \in \sigma$ can make the resulting configuration not simply connected is (i) if it introduces a hole, but this would imply that v had six occupied neighbors in $\sigma \cup V(\Lambda_0)$, which is not allowed; or (ii) if it disconnects the configuration, but that would imply that the neighboring occupied sites to v in $\sigma \cup V(\Lambda_0)$ were disconnected, which we also do not allow. On the other hand, the only way the addition of an occupied site v to σ can violate simple connectivity is if it introduces a cycle that was not present in $\sigma \cup V(\Lambda_0)$. This can only happen if the neighbors of v in $\sigma \cup V(\Lambda_0)$ were either the empty set or if they induced a disconnected graph in $\sigma \cup V(\Lambda_0)$, which we again do not allow. The addition of an occupied site to σ cannot disconnect the resulting configuration. ◀

▶ **Lemma 3** (Irreducibility and Aperiodicity). *The Bridging Occupancy Chain is aperiodic and irreducible. Thus it converges to the stationary distribution π given by Equation 2.*

Proof. We first observe that for every locally simply connected move, the reverse is also valid and locally simply connected. Next, we establish irreducibility of the occupancy chain by showing that from every configuration $\sigma \in \Omega$, we can always identify a site $v \in \sigma$ such that making v unoccupied is a valid move. This would imply that we can always reach the empty configuration via valid moves, and therefore we can connect any pair of configurations $\sigma, \tau \in \Omega$ by first making all the sites in σ unoccupied and then making all the sites in τ occupied by following the reversal of the path from τ to the empty configuration.

Let v be any site in σ that has greatest distance from a site in Λ_0 following edges in $\overline{\Lambda}$ with both endpoints occupied. We will argue that v is locally simply connected. Removing v cannot form a hole because v has greatest distance from Λ_0 and therefore cannot have six occupied neighbors. Likewise, making v unoccupied cannot disconnect σ or there would be occupied sites whose shortest path to $V(\Lambda_0)$ passes through v, contradicting v having greatest shortest distance to $V(\Lambda_0)$. Therefore v is locally simply connected.

Finally, we note that when σ is nearly empty, most choices of v will result in self-loops in the chain, so the chain is aperiodic. Hence, since the chain is ergodic (aperiodic and irreducible), it must have a unique stationary distribution, which is given by Equation 2 due to the Metropolis–Hastings transition probabilities satisfying detailed balance. ◀

Now that we have established the occupancy chain dynamics, it remains to discuss the underlying SOPS dynamics that would govern the behavior of the chain. While the occupancy chain abstraction comes at the cost of obscuring the details of the underlying algorithm that each particle runs, there are indeed local distributed algorithms running independently on constant-memory particles, which we here assume to operate under a sequential scheduler (i.e., where at most one particle is active at any point in time, as in e.g., [4]), that can closely approximate the occupancy chain dynamics. We first note that each iteration of the Bridging Occupancy Chain, and in particular the checks for local simple connectivity within it, can be executed by constant-sized memory agents with only local communication, since the algorithm only depends on the the states of the agents that occupy site v and its immediate neighborhood in Λ.

In the physics agent-based model for the fire-ant bridging biological experiments in [24] that were the original motivation of this work, each particle (ant) performs an independent random walk, biased by the scent gradient as it walks over other particles on the bridge, and biased by both the scent gradient and number of nearest neighbors as it leaves or joins the boundary. The transition probabilities of the occupancy chain are the probabilities with which the bridge structure gains or loses one particle when this particle's walk on the current bridge structure is at stationarity. Moreover, while fine differences in the specific scent gradient affect the motion of the contour, this merely requires that individuals sense, and are acted upon by, the local gradient and does not require them to store and compute with correspondingly fine numbers. Thus, the occupancy chain model is rich enough to capture the long-term behavior of distributed collectives, and can also describe other underlying dynamics because it abstracts details that are irrelevant to our analysis.

3 Equilibrium properties of the occupancy chain

We are now prepared to prove that at most one bridge will form by analyzing the stationary distribution of the occupancy chain. First, we argue that multiple bridges are unlikely if the affinity parameter is sufficiently large. Then, we identify a phase change in the formation of bridges as the scent parameter varies.

3.1 Precluding the formation of multiple bridges

In Section 1.2 we briefly described a configuration with multiple (a, b)-bridges as one that extends out to layer $\Lambda_{\lfloor bh \rfloor}$, retracts back to $\Lambda_{\lfloor ah \rfloor}$, then extends out to $\Lambda_{\lfloor bh \rfloor}$ a second time. We now formally define this, as well as the *depth* of the backtrack for such configurations.

▶ **Definition 4** (Multiple (a, b)-bridges). *For real values $0 < a < b < 1$, a configuration $\sigma \subseteq V(\Lambda)$ has* multiple (a, b)-bridges *if the subgraph of Λ induced by $\bigcup_{k=\lfloor ah \rfloor}^{h} (\sigma \cap V(\Lambda_k))$ contains at least two components that each contain some vertex from $\Lambda_{\lfloor bh \rfloor}$. We call $\epsilon = b - a > 0$ the* depth *of the backtracking.*

To describe the rarity of multiple bridges forming, we consider the event that the configuration has multiple (a, b)-bridges for some pair (a, b) with depth at least $\epsilon > 0$, defined as:

$$\mathsf{MB}_\epsilon = \cup_{a \in (0,1-\epsilon)} \{\sigma \in \Omega : \sigma \text{ has multiple } (a, a + \epsilon)\text{-bridges}\}. \tag{4}$$

We note that this event includes intervals of length greater than ϵ as a configuration with multiple $(a, a + \epsilon')$-bridges for $\epsilon' > \epsilon$ also has multiple $(a, a + \epsilon)$-bridges. Our next result tells us that the probability of MB_ϵ decays exponentially in β and h, so long as β is large enough in terms of the aspect ratio α and depth ϵ.

▶ **Theorem 5** (No Multiple Bridges). *For every $\epsilon > 0$ and every $w \times h$ region Λ with aspect ratio $\alpha = w/h$, there exists a positive number $\beta_0 = \beta_0(\alpha, \epsilon)$ such that, if $\beta > \beta_0$, then the probability $\pi(\mathsf{MB}_\epsilon)$ of multiple bridges with depth ϵ is at most $e^{-\beta h \epsilon/2}$.*

We emphasize that the quantity β_0 in Theorem 5 does not depend on the constant ρ (recall that $n = \lfloor \rho h^2 \rfloor$) and scent parameter η, even though these parameters affect the stationary distribution. Instead, as the proof explains, MB_ϵ is rare because its occurrence requires that the configuration has a relatively long boundary. While the number of configurations in MB_ϵ grows exponentially in the perimeter of the domain Λ, this number is insufficient to compensate for the configurations' low weight when the boundary length penalty is high. Accordingly, for MB_ϵ to be rare, it suffices for β to be large in terms of α, which controls the perimeter, and ϵ, which controls the boundary length when MB_ϵ occurs.

We now state three lemmas that we will use to prove Theorem 5. The first two of these results are due to the technical machinery of layer sequences, which we mentioned in Section 1.2 and which we formally introduce in Section 3.3. We note that, although we their proofs appear later, they do not rely on any earlier results. We state them in terms of the total boundary length and scent components of the Hamiltonian, which we define for a configuration $\sigma \in \Omega$ as

$$B(\sigma) = \sum_{v \in \sigma} B_\sigma(v) \quad \text{and} \quad S(\sigma) = \sum_{v \in \sigma} S(v).$$

▶ **Lemma 6.** *For every $\sigma \in \Omega$, there exists $\tau \in \Omega$ such that $B(\tau) \leq \min\{B(\sigma), 6w + 4h\}$ and $S(\tau) \geq S(\sigma)$.*

Lemma 6 states that in the case where σ has boundary length greater than $6w+4h$, we can always find another configuration τ that has boundary length at most $6w + 4h$ with at least as great a scent component (we can set $\tau = \sigma$ otherwise). This implies that $H(\sigma) \geq H(\tau)$ for all positive values of β and η. Moreover, the boundary length of τ is bounded above in terms of $w + h$, instead of $w \times h$. The proof of Lemma 6 appears in Section 3.3.

If σ has multiple bridges, then we can further guarantee that there is a configuration τ with a boundary that is strictly shorter by an amount proportional to the depth of the bridges. This is the content of Lemma 7, which we prove in Appendix A.

▶ **Lemma 7.** *For every $\epsilon > 0$ and $\sigma \in \mathsf{MB}_\epsilon$, there exists $\tau \in \Omega$ such that $B(\sigma) \geq B(\tau) + \epsilon h$.*

Together, Lemmas 6 and 7 imply that a configuration σ with multiple bridges can be mapped to a configuration τ that has far greater probability under π because $H(\sigma) - H(\tau) \geq \beta\epsilon h$. This observation is one part of the proof of Theorem 5. The other part is a bound on the number of configurations with a given boundary length.

▶ **Lemma 8.** *The number of configurations with a boundary length of ℓ is at most $2^{\ell+2w}$.*

Proof. Each configuration $\sigma \in \Omega$ of boundary length ℓ is fully described by a collection of self-avoiding paths of total length ℓ on the dual lattice. Each of these paths must start and end on either the top or bottom of Λ, giving $2w$ possible starting points. There are at most 2^{2w} ways to label each starting point as used or unused, and no more than 2^ℓ ways to draw these paths given known starting points as there are at most two directions on the dual lattice to take each step of the path, giving an upper bound of $2^{\ell+2w}$. ◀

With these three lemmas, we are now prepared to prove our first main theorem:

Proof of Theorem 5. We aim to upper bound the probability that the event MB_ϵ occurs, using the fact that we can map the configurations it comprises to configurations with shorter boundaries. More precisely, by Lemma 7, for every $\sigma \in \mathsf{MB}_\epsilon$, there exists $\tau_\sigma \in \Omega$ such that

$$B(\sigma) - B(\tau_\sigma) \geq \epsilon h. \tag{5}$$

In fact, we can assume that τ_σ further satisfies

$$S(\sigma) \leq S(\tau_\sigma) \quad \text{and} \quad B(\tau_\sigma) \leq 6w + 4h \tag{6}$$

because, if it did not, then we could use Lemma 6 to map τ_σ to a configuration that satisfies all three bounds in its place. We partition MB_ϵ according to the difference in (5):

$$\mathsf{MB}_\epsilon = \cup_{k \geq \epsilon h} \mathsf{MB}_{\epsilon,k}, \quad \text{where} \quad \mathsf{MB}_{\epsilon,k} = \{\sigma \in \mathsf{MB}_\epsilon : B(\sigma) - B(\tau_\sigma) = k\}. \tag{7}$$

The probability of each $\sigma \in \mathsf{MB}_{\epsilon,k}$ is exponentially small in k, due to the first bound in (6):

$$\pi(\sigma) \leq \frac{\pi(\sigma)}{\pi(\tau_\sigma)} = e^{-\beta(H(\sigma)-H(\tau_\sigma))} \leq e^{-\beta(B(\sigma)-B(\tau_\sigma))} = e^{-\beta k}. \tag{8}$$

This bound is useful because $|\mathsf{MB}_{\epsilon,k}|$ is at most exponentially large in k. Indeed, due to the second bound in (6), every $\sigma \in \mathsf{MB}_{\epsilon,k}$ has boundary length of at most

$$B(\sigma) = B(\tau_\sigma) + k \leq 6w + 4h + k.$$

There are at most $2^{2w+\ell}$ configurations with a boundary length of exactly ℓ (Lemma 8), hence the number of configurations in $\mathsf{MB}_{\epsilon,k}$ is at most

$$|\mathsf{MB}_{\epsilon,k}| \leq |\{\sigma \in \Omega : B(\sigma) \leq 6w + 4h + k\}| \leq \sum_{\ell \leq 6w+4h+k} 2^{2w+\ell} \leq 4^{7w+4h+k}. \tag{9}$$

We combine the partition of MB_ϵ with the estimates of $\pi(\sigma)$ and $|\mathsf{MB}_{\epsilon,k}|$ to conclude that

$$\pi(\mathsf{MB}_\epsilon) \overset{(7)}{=} \sum_{k \geq \epsilon h} \sum_{\sigma \in \mathsf{MB}_{\epsilon,k}} \pi(\sigma) \overset{(8)}{\leq} \sum_{k \geq \epsilon h} |\mathsf{MB}_{\epsilon,k}| e^{-\beta k} \overset{(9)}{\leq} \sum_{k \geq \epsilon h} 4^{7w+4h+k} e^{-\beta k}.$$

Since $w = \alpha h$, this bound is at most $e^{-\beta h \epsilon/2}$ when β is large enough in terms of α and ϵ. ◀

3.2 Conditions for bridge formation

In this section, we show that whether a bridge to the food forms depends on the balance between the scent and boundary length components of the Hamiltonian. Recall that η controls each particle's desire to move toward the food relative to its desire for more neighbors.

The competition between scent and boundary length is richest when there are not too few or too many particles in the system. When there are too few particles, no bridges can form; when there are too many particles, "bridges" can form without incurring a high boundary cost, due to the periodic boundary conditions. Accordingly, we assume that the number of particles n is $\lfloor \rho h^2 \rfloor$, for a constant ρ that satisfies $1/2 < \rho < \alpha - 2$, in terms of the aspect ratio $\alpha = w/h$ of the domain Λ. Implicitly, we require that α is greater than 2.5.

We write the scent intensity as a function $S_h : [h] \to \mathbb{R}_{\geq 0}$, where $S_h(k)$ represents the intensity of any particle on layer k. Previously, we only assumed that S_h was a nondecreasing function (i.e., nonincreasing in the distance from the food), and showed that as long as β is sufficiently high, at most one bridge forms. Now we assume additional light conditions on S_h, which we believe will apply for most "natural" definitions of the scent intensity function. We use the term *nondecelerating functions* to refer to this broad class of functions.

▶ **Definition 9.** *For a function $S_h : [h] \to \mathbb{R}_{\geq 0}$, let $\Delta S_h(k) = S_h(k+1) - S_h(k)$ denote the discrete derivative of S_h at $k \in [h-1]$. We say that S_h is* nondecelerating *if $S_h(1) = 0$, $\Delta S_h(k) \geq 0$ for all $k \in [h-1]$, and $\Delta(\Delta S_h)(k) \geq 0$ for all $k \in [h-2]$.*

Assuming that the scent function is nondecelerating, our second main result identifies a phase change in the probability that no bridge reaches the food. We define this event as

$$\mathsf{NB} = \{\sigma \in \Omega : \sigma \cap V(\Lambda_h) = \emptyset\}.$$

The phases correspond to ranges of β and η that depend on the *column scent* $\varphi = \sum_{k=1}^{h} S_h(k)$, which represents the sum of the scent intensities over a single "column" of Λ. We consider the case when $\varphi > 0$ is a constant with respect to h, the aspect ratio satisfies $\alpha > 2.5$, and the constant ρ belongs to $(0.5, \alpha - 2)$. We define the phases in terms of the quantities

$$\beta_1 = \frac{2\rho + 3 + 4\alpha}{2\rho - 1}\log 2, \quad \beta_2 = \left(1 + \frac{\alpha}{2}\right)\log 2, \quad \text{and} \quad \eta_1 = \frac{4}{\varphi}\left(1 + \frac{1}{\rho}\right),$$

as well as $\eta_2 = \eta_2(\beta)$, given by

$$\eta_2 = \frac{1}{\varphi}\min\left\{2\left(1 - \frac{\log 2}{\beta}\right), \frac{4}{1+\rho}\left(1 - \left(1 + \frac{\alpha}{2}\right)\frac{\log 2}{\beta}\right)\right\}.$$

Note that both η_1 and η_2 are proportional to $1/\varphi$, and η_2 is strictly positive when $\beta > \beta_2$.

▶ **Theorem 10** (Phase Change). *Suppose that $1/2 < \rho < \alpha - 2$ and let S_h be nondecelerating.*

(i) (At Least One Bridge) *If $\beta > \beta_1$ and $\eta > \eta_1$, then there exist positive numbers $c_1 = c_1(\beta, \eta)$ and $h_1 = h_1(\beta, \eta)$ such that $\pi(\mathsf{NB}) \le e^{-c_1 h}$, for all $h \ge h_1$.*

(ii) (No Bridge) *If $\beta > \beta_2$ and $\eta < \eta_2(\beta)$, then there exist positive numbers $c_2 = c_2(\beta, \eta)$ and $h_2 = h_2(\beta, \eta)$ such that $\pi(\mathsf{NB}) \ge 1 - e^{-c_2 h}$, for all $h \ge h_2$.*

Since η_1 and η_2 are proportional to $1/\varphi$, an immediate corollary of Theorem 10 is that, if the scent is especially strong or especially weak, in the sense that the asymptotic growth of the column scent as $h \to \infty$ satisfies $\varphi = \omega_h(1)$ or $\varphi = o_h(1)$, then bridge formation is either likely or unlikely, independently of η.

▶ **Corollary 11.** *Suppose that $1/2 < \rho < \alpha - 2$ and S_h is nondecelerating.*

(i) (Strong Scent) *If $\beta > \beta_1$ and $\varphi = \omega_h(1)$, then there exist positive numbers $c_3 = c_3(\beta)$ and $h_3 = h_3(\beta)$ such that $\pi(\mathsf{NB}) \le e^{-c_3 h}$, for all $h \ge h_3$.*

(ii) (Weak Scent) *If $\beta > \beta_2$ and $\varphi = o_h(1)$, then there exist positive numbers $c_4 = c_4(\beta)$ and $h_4 = h_4(\beta)$ such that $\pi(\mathsf{NB}) \ge 1 - e^{-c_4 h}$, for all $h \ge h_4$.*

Note Corollary 11(ii) holds without the assumption that S_h is nondecelerating. In fact, it holds when S_h is nonnegative because, when $\varphi = o_h(1)$, the contribution of boundary length to the Hamiltonian dominates the scent component, making fine properties of S_h irrelevant.

The proof of Theorem 10 relies on three lemmas. Informally, the first says that a subset of configurations Ω^{bad} has exponentially small probability in h if every $\sigma \in \Omega^{\mathrm{bad}}$ can be mapped to a configuration with a Hamiltonian that is smaller by roughly $h + B(\sigma)$. The proof, which can be found in the full version of the paper [17], is similar to that of Theorem 5.

▶ **Lemma 12.** *Fix an aspect ratio α, as well as δ and ϵ, all positive numbers. Suppose that Ω^{bad} is a subset of configurations such that, for every $\sigma \in \Omega^{\mathrm{bad}}$, there exists $\tau_\sigma \in \Omega$ satisfying*

$$H(\sigma) - H(\tau_\sigma) \ge \epsilon h + \delta B(\sigma) + O_h(1),$$

where the function implicit in $O_h(1)$ does not depend on σ. Then, for every

$$\beta > \beta_0(\delta, \epsilon) := \max\left\{\frac{\log 2}{\delta}, \frac{2\alpha \log 2}{\epsilon}\right\},$$

there exist $c = c(\beta) > 0$ and $h_0 = h_0(\beta) > 0$ such that $\pi(\Omega^{\mathrm{bad}}) \le e^{-ch}$ for all $h \ge h_0$.

The next lemma states that if a configuration does not form a bridge, then it is possible to identify a configuration with a Hamiltonian that is strictly smaller as a function of h and the boundary length of the original configuration.

▶ **Lemma 13.** *If $\beta > \beta_1$ and $\eta > \eta_1$, then there exist positive values $\delta_1 = \delta_1(\rho, \alpha)$ and $\epsilon_1 = \epsilon_1(\rho, \alpha)$ such that $\beta_1 \geq \beta_0(\delta_1, \epsilon_1)$ and, for every $\sigma \in \mathsf{NB}$, there exists $\tau_\sigma \in \Omega$ satisfying*

$$H(\sigma) - H(\tau_\sigma) \geq \epsilon_1 h + \delta_1 B(\sigma) + O_h(1).$$

The proof involves a multi-stage transformation of the original configuration's layer sequence (Appendix A). The same is true of the next result, which instead compares configurations that reach Λ_h to the empty configuration $\emptyset$, which has $H(\emptyset) = 0$.

▶ **Lemma 14.** *If $\beta > \beta_2$ and $\eta < \eta_2(\beta)$, then there exist positive values $\delta_2 = \delta_2(\eta, \varphi, \rho, \alpha)$ and $\epsilon_2 = \epsilon_2(\eta, \varphi, \rho, \alpha)$ such that $\beta_2 \geq \beta_0(\delta_2, \epsilon_2)$ and every $\sigma \notin \mathsf{NB}$ satisfies*

$$H(\sigma) - H(\emptyset) \geq \epsilon_2 h + \delta_2 B(\sigma) + O_h(1).$$

Theorem 10 follows directly from the preceding lemmas.

Proof of Theorem 10. Parts (i) and (ii) follow from two applications of Lemma 12, one to $\Omega^{\text{bad}} = \mathsf{NB}$ and another to $\Omega^{\text{bad}} = \mathsf{NB}^c$, which are justified by Lemmas 13 and 14. ◀

3.3 Layer sequences

The proofs of our main results rely on mappings of configurations to ones with more desirable properties (Lemmas 6, 7, 13, and 14). For example, Lemma 6 states that every configuration can be mapped to a second configuration with at least as favorable boundary length and scent components, and which has a boundary length of $O(w + h)$. The purpose of this section is to introduce layer sequences, the key technical idea underlying these lemmas, and to demonstrate their use by proving Lemma 6.

The layer sequence $\overline{N} = (n_k)_{k=1}^h$ of a (possibly non-valid) configuration $\sigma \subseteq V(\Lambda)$ counts the number $n_k = n_k(\sigma)$ of its elements in each layer of Λ:

$$n_k(\sigma) = |\sigma \cap V(\Lambda_k)|.$$

The statements that we will make in this section will apply not only to configurations in Ω, but a broader class $\overline{\Omega}$ of configurations $\sigma \subseteq V(\Lambda)$ that have layer sequences that have no fully occupied layer (k where $n_k(\sigma) = w$) after one that is not fully occupied, and no fully unoccupied layer (k where $n_k(\sigma) = 0$) before one that is not fully unoccupied:

$$\begin{aligned}\overline{\Omega} = \big\{\sigma \subseteq V(\Lambda) :\ & (n_{k+1}(\sigma) = w \implies n_k(\sigma) = w) \text{ and} \\ & (n_k(\sigma) = 0 \implies n_{k+1}(\sigma) = 0) \text{ for all } k \in [h-1]\big\}.\end{aligned}$$

Notably, configurations in Ω must have this property as they are simply connected, so we can say that $\Omega \subseteq \overline{\Omega}$. Referencing this property, for $0 \leq R_{bot} \leq R_{top} \leq h$, we can denote by $\mathcal{L}^{R_{top}, R_{bot}}$ the set of layer sequences that are fully occupied from layers 1 to R_{bot}, partially occupied from layers $R_{bot} + 1$ to R_{top}, and fully unoccupied for the remaining layers.

Even though the layer sequence of a configuration discards information about the configuration's boundary, we can still use it to obtain bounds on the boundary length (our definition of boundary length extends to configurations in $\overline{\Omega}$). This is the content of the next two results. To state them, we denote by σ_M the part of a configuration $\sigma \in \overline{\Omega}$ in layers 1 through M:

$$\sigma_M = \cup_{k=1}^M (\sigma \cap V(\Lambda_k)).$$

▶ **Lemma 15** (Boundary length increment). *Let $0 \leq R_{bot} \leq R_{top} \leq h$ and let $\sigma \in \overline{\Omega}$ have layer sequence $\overline{N} \in \mathcal{L}^{R_{top},R_{bot}}$. For integers M where $\max\{R_{bot}, 1\} \leq M \leq \min\{R_{top} - 1, h - 2\}$, the truncation σ_{M+1} has boundary length satisfying $B(\sigma_{M+1}) \geq B(\sigma_M) + D_M$, where*

$$D_M = 2 + 2 \max\{n_{M+1} - n_M + 1, 0\} + 2 \max\{n_{M+1} - n_M - 1, 0\}.$$

If $M = h - 1$ instead, then we have $B(\sigma_{M+1}) = B(\sigma) \geq B(\sigma_M) + D_M - 2n_h$. Furthermore, the bound is tight in the sense that, for any $\overline{N} \in \mathcal{L}^{R_{top},R_{bot}}$, there exists $\tau \in \Omega$ (not $\overline{\Omega}$) with layer sequence $\overline{N}$ which satisfies $B(\tau_{M+1}) \leq B(\tau_M) + D_M$ for every such M.

The proof of Lemma 15, which is self-contained, appears in Appendix A. Lemma 15 implies a lower bound on a configuration's boundary length in terms of its layer sequence.

▶ **Lemma 16** (Boundary length lower bounds). *Suppose that $\overline{N} \in \mathcal{L}^{R_{top},R_{bot}}$ where $0 \leq R_{bot} \leq R_{top} \leq h$ and let $R_{min} = \max\{R_{bot}, 1\}$. For each $M \in \{R_{min}, R_{min} + 1, \ldots, R_{top}\}$, we define*

$$\overline{B}_M(\overline{N}) = \overline{B}_{R_{min}}(\overline{N}) + \sum_{k=R_{min}}^{M-1} D_k, \quad \text{where} \quad \overline{B}_{R_{min}}(\overline{N}) = \begin{cases} 2n_1 + 2 & \text{if } R_{bot} = 0, \\ 2w & \text{if } R_{bot} \geq 1. \end{cases}$$

Then, for any configuration $\sigma \in \overline{\Omega}$ with layer sequence $\overline{N}$, for any $M \in \{R_{min}, R_{min} + 1, \ldots, R_{top}\}$, the truncation σ_M has boundary length satisfying

$$B(\sigma_M) \geq \begin{cases} \overline{B}_M(\overline{N}) & \text{if } M < h, \\ \overline{B}_M(\overline{N}) - 2n_h & \text{if } M = h. \end{cases}$$

Furthermore, this bound is tight in the sense that for any layer sequence $\overline{N} \in \mathcal{L}^{R_{top},R_{bot}}$, there exists a configuration $\tau \in \Omega$ with layer sequence $\overline{N}$ where $B(\tau_{R_{bot}+1}) = \overline{B}_{R_{bot}+1}(\overline{N})$ and $B(\tau_{M+1}) \leq B(\tau_M) + D_M$ for each $M \in \{R_{min}, R_{min} + 1, \ldots, R_{top}\}$.

Proof of Lemma 16. This follows from Lemma 15 by induction on M, with $M = R_{\min}$ as the base case. ◀

We use the two preceding facts about layer sequences to prove Lemma 6. The proof features the quantity $\overline{S}(\overline{N})$, which is the scent intensity common to all configurations with layer sequence $\overline{N}$.

Proof of Lemma 6. We aim to show that, for an arbitrary configuration $\sigma \in \Omega$, there exists $\tau \in \Omega$ such that $B(\tau) \leq B(\sigma)$, $S(\tau) \geq S(\sigma)$, and $B(\tau) \leq 6w + 4h$. The idea of the proof is to show that, if there are layers k_1 and k_2 such that the layer sequence $\overline{N}$ of σ satisfies

$$k_2 \geq k_1 + 2, \quad n_{k_1+1} - n_{k_1} \geq 2, \quad \text{and} \quad n_{k_2+1} - n_{k_2} \geq 2,$$

then it is possible to "promote" one of the occupied sites to a higher layer, resulting in a new layer sequence $\overline{N}^+ = (n_k^+)_{k\in[h]} \in \mathcal{L}^{R_{bot}^+,R_{top}^+}$, in such a way that $\overline{B}_h(\overline{N}^+) \leq \overline{B}_h(\overline{N})$. Moving an occupied site to a higher layer clearly also ensures that $\overline{S}(\overline{N}^+) \geq \overline{S}(\overline{N})$, because S_h is nonincreasing with distance from the food. We define $\overline{N}^+$ by removing an occupied site from layer $k_1 + 1$ and adding it to layer k_2. In other words, for each $k \in [h]$, we define: $n_k^+ = n_k - 1$ if $k = k_1 + 1$; $n_k^+ = n_k + 1$ if $k = k_2$; and $n_k^+ = n_k$ otherwise. Note that, like R_{bot}, R_{bot}^+ is at most k_1.

To compute the difference between $\overline{B}_h(\overline{N}^+)$ and $\overline{B}_h(\overline{N})$, we look at the terms affected by the change. For each k, denote by D_k and D_k^+ the terms of the sums of $\overline{B}_h(\overline{N})$ and $\overline{B}_h(\overline{N}^+)$. We observe that $D_{k_1}^+ = D_{k_1} - 4$ and $D_{k_2}^+ = D_{k_2} - 4$, and the differences $D_{k_1+1}^+ - D_{k_1+1}$ and $D_{k_2-1}^+ - D_{k_2-1}$ are at most 8 when $k_1 + 1$ and $k_2 - 1$ are equal, and at most 4 when they are not. This implies that $\overline{B}_h(\overline{N}^+) \leq \overline{B}_h(\overline{N}) - 8 + 8 = \overline{B}_h(\overline{N})$.

We repeat this process of promoting occupied sites to find a layer sequence $\overline{N}^* = (n_k^*)_{k\in[h]}$ such that $\overline{B}_h(\overline{N}^*) \leq \overline{B}_h(\overline{N})$, $\overline{S}(\overline{N}^*) \geq \overline{S}(\overline{N})$, and $n_{k+1}^* - n_k^* \leq 1$ for all but at most two values of k. Moreover, if there are two such values of k, they must be consecutive integers. This implies the upper bound

$$\overline{B}_h(\overline{N}^*) \leq \overline{B}_{R_{bot}^*+1}(\overline{N}) + 4(R_{top}^* - R_{bot}^*) + 4w \leq 6w + 4h.$$

By Lemma 16, there exists $\tau \in \Omega$ with layer sequence $\overline{N}^*$ and $B(\tau) \leq \overline{B}_h(\overline{N}^*)$, hence $B(\tau) \leq 6w + 4h$. ◀

4 Conclusion

We define a simple SOPS model of bridging based on the particles' affinity for more neighbors, which results in more "robust" bridges, and a bias toward the food. We show that the emergence of a single bridge in collective systems is a statistical inevitability, requiring no central coordination. The novelty of our strategy is based on defining and analyzing a much simpler occupancy chain that abstracts the specific local dynamics of the particles and looks at the evolution of the contour indicating the system's occupied sites. We intentionally kept the occupancy chain as simple as possible to show the generality of the single bridging emergent phenomena to highlight the connections to similar statistical physics models.[2] Moreover, the occupancy chain that evolves according to a Hamiltonian defined only by contour length and scent gradient is simple enough to provide rigorous proofs using tools from statistical physics.

We see several directions for future work: First, we believe that we can extend our results to other models inspired by biological systems and beyond, such as by including ants retreat from the food after they are fed, by using a more sophisticated occupancy chain that models rafts by allowing a collection of ants to move together in a single move, or by including more specifics of man-made swarm robotics systems. Our simulations show that such variants do not significantly change our findings. Second, the occupancy chain abstracts away information about the motion of individual particles, and allows us to gain a more direct means of analyzing their collective behavior: We expect that such abstractions will help understanding the collective behavior of many other programmable matter systems. Third, occupancy Markov chains have the potential for impact in statistical physics, by allowing one to relax the need for precisely estimating surface tension of contours, potentially enabling a better formal understanding of fixed magnetization spin systems [8], and collectives arising in swarm robotics responding to directed external stimuli [12]. Lastly, while simulations suggest that both the occupancy chain and the underlying agent-based simulations converge in polynomial time, we do not have a formal bound on the mixing time of either. Future work can relax the connectivity restriction on valid configurations, which may make it easier to derive such bounds, but ant experiments suggest typically the majority do stay connected.

[2] The techniques we use also apply to other geometries, including the square lattice and other planar graphs with or without periodic boundary conditions.

It would also be helpful to derive general bounds relating the mixing times of an occupancy chain with the underlying dynamics for a general class of models, perhaps building on similar decomposition theorems [14].

References

1 M. Andrés Arroyo, S. Cannon, J.J. Daymude, D. Randall, and A.W. Richa. A stochastic approach to shortcut bridging in programmable matter. *Natural Computing*, 17(4):723–741, 2018. doi:10.1007/S11047-018-9714-X.

2 S. Camazine, K.P. Visscher, J. Finley, and S.R. Vetter. House-hunting by honey bee swarms: Collective decisions and individual behaviors. *Insectes Sociaux*, 46(4):348–360, 1999.

3 S. Cannon, J.J. Daymude, C. Gökmen, D. Randall, and A.W. Richa. A local stochastic algorithm for separation in heterogeneous self-organizing particle systems. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX-/RANDOM)*, pages 54:1–54:22, 2019.

4 S. Cannon, J.J. Daymude, D. Randall, and A.W. Richa. A Markov chain algorithm for compression in self-organizing particle systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 279–288, 2016.

5 J. Daudelin, G. Jing, T. Tosun, M. Yim, H. Kress-Gazit, and M. Campbell. An integrated system for perception-driven autonomy with modular robots. *Science Robotics*, 3(23):eaat4983, 2018. doi:10.1126/scirobotics.aat4983.

6 J.J. Daymude, A.W. Richa, and C. Scheideler. The canonical amoebot model: Algorithms and concurrency control. *CoRR*, abs/2105.02420, 2021. arXiv:2105.02420.

7 Z. Derakhshandeh, S. Dolev, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. Brief announcement: Amoebot - a new model for programmable matter. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 220–222, 2014.

8 R.L. Dobrushin, R. Kotecky, and S.B. Shlosman. *The Wulff Construction: a Global Shape from Local Interactions.* American Mathematical Society, Providence, 1992.

9 S. Greenberg, D. Randall, and A.P. Streib. Sampling biased monotonic surfaces usijng exponential metrics. *Combinatorics, Probability and Computing*, 29:672–697, 2020. doi:10.1017/S0963548320000188.

10 R. Jeanson, C. Rivault, J.L. Deneubourg, S. Blanco, R. Fournier, C. Jost, and G. Theraulaz. Self-organized aggregation in cockroaches. *Animal Behaviour*, 69(1):169–180, 2005.

11 E. Karzbrun, A.M. Tayar, V. Noireaux, and R.H. Bar-Ziv. Programmable on-chip DNA compartments as artificial cells. *Science*, 345(6198):829–832, 2014. doi:10.1126/science.1255550.

12 S. Li, B. Dutta, S. Cannon, J.J. Daymude, R. Avinery, E. Aydin, A.W. Richa, D.I. Goldman, and D. Randall. Programming active cohesive granular matter with mechanically induced phase changes. *Science Advances*, 7(17):eabe8494, 2021. doi:10.1126/sciadv.abe8494.

13 A. Lioni, C. Sauwens, G. Theraulaz, and J.-L. Deneubourg. Chain formation in Oecophylla longinoda. *Journal of Insect Behavior*, 14(5):679–696, 2001.

14 N. Madras and D. Randall. Markov chain decomposition for convergence rate analysis. *Annals of Applied Probability*, 12(2):581–606, 2002.

15 N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

16 N.J. Mlot, C.A. Tovey, and D.L. Hu. Fire ants self-assemble into waterproof rafts to survive floods. *Proceedings of the National Academy of Sciences*, 108(19):7669–7673, 2011.

17 S. Oh, J.L. Briones, J. Calvert, N. Egan, D. Randall, and A.W. Richa. Single bridge formation in self-organizing particle systems. ArXiv preprint *http://www.arxiv.org/abs/2408.10830*, 2024.

18 N.T. Ouellette and D.M. Gordon. Goals and limitations of modeling collective behavior in biological systems. *Frontiers in Physics*, 9, 2021. doi:10.3389/fphy.2021.687823.

19 C.R. Reid, M.J. Lutz, S. Powell, A.B. Kao, I.D. Couzin, and S. Garnier. Army ants dynamically adjust living bridges in response to a cost–benefit trade-off. *Proceedings of the National Academy of Sciences*, 112(49):15113–15118, 2015.

20 E. Şahin. Swarm robotics: From sources of inspiration to domains of application. In *Swarm Robotics*, pages 10–20, 2005.

21 W. Savoie, S. Cannon, J.J. Daymude, R. Warkentin, S. Li, A.W. Richa, D. Randall, and D. I. Goldman. Phototactic supersmarticles. *Artificial Life and Robotics*, 23(4):459–468, 2018. doi:10.1007/S10015-018-0473-7.

22 B. Wei, M. Dai, and P. Yin. Complex shapes self-assembled from single-stranded DNA tiles. *Nature*, 485(7400):623–626, 2012. doi:10.1038/nature11075.

23 A. Yadav. Stochastic maze solving under the geometric amoebot model. Master's thesis, Rutgers University, 2021. URL: https://rucore.libraries.rutgers.edu/rutgers-lib/65707/.

24 H. Zeng, J. Briones, R. Avinery, S. Li, A. Richa, D. Goldman, and T. Sasaki. Fire ant pontoon bridge: a self-assembled dynamic functional structure. In *Integrative and Comparative Biology*, volume 62, pages S341–S342, 2023.

A Additional details of the technical lemmas

In this appendix, we further develop the properties and applications of layer sequences, introduced in Section 3.3. First, in Section A.1, we collect the proofs of some inputs to the proof of Theorem 5. Then, in Section A.2, we discuss the main ideas behind the inputs to the proof of Theorem 10.

A.1 Further inputs to Theorem 5

Proof of Lemma 15. For $\sigma \in \overline{\Omega}$, we look at the change in total boundary length when we add layer $M+1$ of σ to σ_M. Let C_{M+1} be the number of components in the restriction of σ to only layer $M+1$.

As layer $M+1$ is not completely filled, there are exactly $n_{M+1} - C_{M+1}$ edges between occupied sites within layer $M+1$. Thus, the number of edges of Λ with exactly one endpoint at an occupied site in layer $M+1$ is equal to $6n_{M+1} - 2(n_{M+1} - C_{M+1}) = 4n_{M+1} + 2C_{M+1}$. Some of the edges are shared with occupied sites on layer M. Denoting by E_{M+1} the number of edges between layers M and $M+1$, we can see that adding layer $M+1$ to σ_M adds $4n_{M+1} + 2C_{M+1} - 2E_{M+1}$ to the boundary length of σ_{M+1}.

We compute an upper bound for E_{M+1} by counting the number of occupied sites in layer $M+1$ with at least one and with at least two edges to layer M respectively. Layer M, with C_M components, provides $n_M + C_M$ occupied sites in layer $M+1$ with at least one edge to layer M, and $n_M - C_M$ occupied sites in layer $M+1$ with two edges to layer M. A site in layer $M+1$ cannot have more than two edges to layer M. This gives an upper bound of $\min\{n_M + c_M, n_{M+1}\}$ for the former, and $\min\{n_M - c_M, n_{M+1}\}$ for the latter. This gives us the following lower bound:

$$\begin{aligned}
&B(\sigma_{M+1}) - B(\sigma_M)\\
&\qquad \geq 4n_{M+1} + 2C_{M+1} - 2\min\{n_M + C_M, n_{M+1}\} - 2\min\{n_M - C_M, n_{M+1}\}\\
&\qquad \geq 2C_{M+1} + 2\max\{n_{M+1} - n_M - C_M, 0\} + 2\max\{n_{M+1} - n_M + C_M, 0\}\\
&\qquad \geq 2 + 2\max\{n_{M+1} - n_M - 1, 0\} + 2\max\{n_{M+1} - n_M + 1, 0\}.
\end{aligned}$$

The last inequality holds because $C_{M+1} \geq 1$ and $\max\{n_{M+1} - n_M - C_M, 0\} + \max\{n_{M+1} - n_M + C_M, 0\} \geq \max\{n_{M+1} - n_M - 1, 0\} + \max\{n_{M+1} - n_M + 1, 0\}$ for any value of $n_{M+1} - n_M$, as long as $C_M \geq 1$ (which is true as $\sigma \in \overline{\Omega}$). To show the second statement of the lemma, we can construct σ layer by layer, where each layer from $\max\{1, R_{bot}\}$ to R_{top} has exactly one component and M_{M+1} is maximized for each value of M. ◀

Proof of Lemma 7. In this proof, we will use $(x)_+$ to denote $\max\{x, 0\}$ for $x \in \mathbb{R}$. Fix $\epsilon \geq 1/h$ and a configuration $\sigma \in \mathsf{MB}_\epsilon$. By definition (4), there is an $a \in (0,1)$ such that σ has multiple $(a, a+\epsilon)$-bridges. Set $M = \lfloor ah \rfloor$, and let $\Lambda_{\geq M}$ represent the set of sites on layers M to h. We can divide the elements of σ within $\Lambda_{\geq M}$ into connected components $V_1, V_2, \ldots, V_P$ for some integer P. Denote by V_{base} the elements of σ not in $\Lambda_{\geq M}$. For each $j \in \{1, 2, \ldots, P\}$, we denote $\sigma^{(j)} = V_j \cup V_{\text{base}}$ which we note will be a configuration in $\overline{\Omega}$, and let $\overline{N}^{(j)} = (n_k^{(j)})_{k \in \{0,1,\ldots,h\}}$ be its layer sequence.

By applying Lemma 15, the increase in boundary length when adding the sites V_j to the restriction of σ to the layers $1, 2, \ldots, M-1$ must be at least $\overline{B}^{(j)}$, where

$$\overline{B}^{(j)} = \sum_{k=M}^{R_{top}^{(j)}-1} \left(2 + 2\left(n_{k+1}^{(j)} - n_k^{(j)} + 1\right)_+ + 2\left(n_{k+1}^{(j)} - n_k^{(j)} - 1\right)_+\right),$$

and where $R_{top}^{(j)}$ denotes the topmost non-empty layer of $\sigma^{(j)}$. We can thus give the following lower bound for the boundary length of σ:

$$B(\sigma) \geq \overline{B}_M(\overline{N}) + \sum_{j=1}^{P} \overline{B}^{(j)}.$$

Denoting by $\overline{N} = (n_k)_{k \in \{0,1,\ldots,h\}}$ the layer sequence of σ itself, by Lemma 16, there exists a configuration τ with layer sequence $\overline{N}$ where:

$$B(\tau) = \overline{B}_M(\overline{N}) + \sum_{k=M}^{R_{top}-1} \left(2 + 2\left(n_{k+1} - n_k + 1\right)_+ + 2\left(n_{k+1} - n_k - 1\right)_+\right).$$

Without loss of generality, we may assume that $R_{top} = R_{top}^{(1)} \geq R_{top}^{(2)} \geq \ldots \geq R_{top}^{(P)}$.

$$\begin{aligned} B(\sigma) - B(\tau) &\geq \sum_{j=1}^{P} \overline{B}^{(j)} - \sum_{k=M}^{R_{top}} \left(2 + 2\left(n_{k+1} - n_k + 1\right)_+ + 2\left(n_{k+1} - n_k - 1\right)_+\right) \\ &= -\sum_{j=2}^{P} \sum_{k=M}^{R_{top}^{(j)}} 2 + 2\sum_{k=M}^{R_{top}^{(j)}} \left(\sum_{\substack{j \in [P] \\ k \leq R_{top}^{(j)}-1}} \left(n_{k+1}^{(j)} - n_k^{(j)} + 1\right)_+ - \left(n_{k+1} - n_k + 1\right)_+ \right) \\ &\qquad + 2\sum_{k=M}^{R_{top}^{(j)}} \left(\sum_{\substack{j \in [P] \\ k \leq R_{top}^{(j)}-1}} \left(n_{k+1}^{(j)} - n_k^{(j)} - 1\right)_+ - \left(n_{k+1} - n_k - 1\right)_+ \right). \end{aligned}$$

The second and third terms of the above expression are non-negative as for $I \in \{1, -1\}$, we have

$$\sum_{\substack{j\in[P]\\ k\le R_{top}^{(j)}-1}} \left(n_{k+1}^{(j)} - n_k^{(j)} + I\right)_+ - (n_{k+1} - n_k + I)_+$$
$$= \sum_{\substack{j\in[P]\\ k\le R_{top}^{(j)}}} \left(n_{k+1}^{(j)} - n_k^{(j)} + I\right)_+ - \Big(\sum_{\substack{j\in[P]\\ k\le R_{top}^{(j)}}} (n_{k+1} - n_k) + I\Big)_+ \ge 0.$$

This allows us to conclude that

$$B(\sigma) - B(\tau) \ge \sum_{j=2}^{P} \sum_{k=M}^{R_{top}^{(j)}} 2 = 2\sum_{j=2}^{P} (R_{top}^{(j)} - M + 1). \tag{10}$$

If σ has multiple $(a, a+\epsilon)$-bridges, we must have $P \ge 2$ and $R_{top}^{(2)} - M + 1 \ge \epsilon h$, and so according to (10) we must have $B(\sigma) \ge B(\tau) + 2\epsilon h$. ◀

A.2 Further inputs to Theorem 10

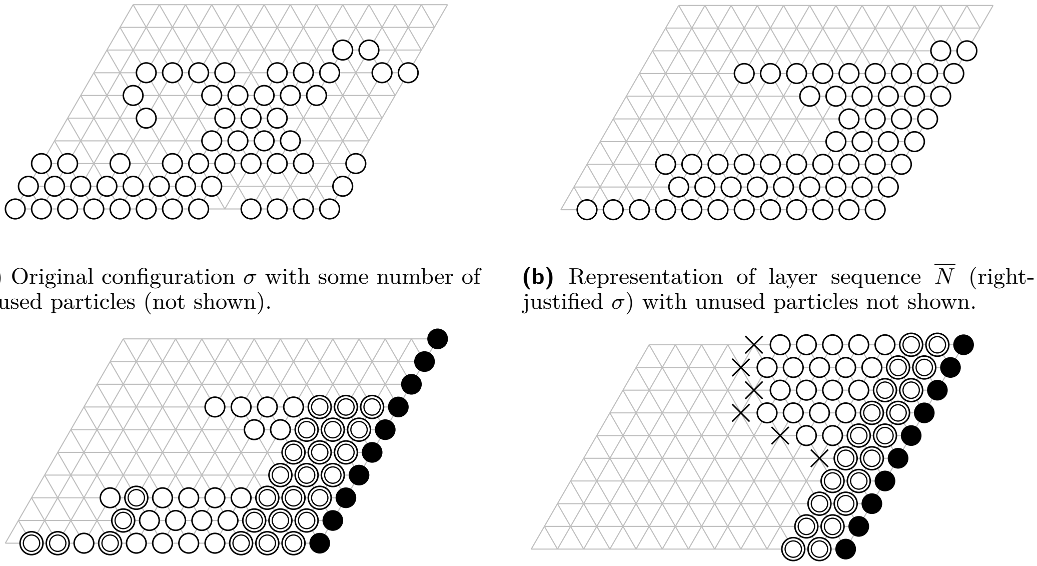
In this section, we present the main ideas behind Lemmas 13 and 14, which we prove in the full version of the paper [17]. These lemmas arise from the analysis of a transformation of layer sequences which gives a significant improvement (more negative) in the Hamiltonian when a configuration without a bridge to the food is extended to reach the food, so long as the scent parameter η is sufficiently large.

Let σ be an arbitrary configuration with layer sequence $\overline{N} = (n_k)_{k\in[h]}$. Suppose that $\overline{N} \in \mathcal{L}^{0,D}$, where $D \le h-1$ is the highest occupied layer in σ. We will transform $\overline{N}$ into a new layer sequence $\overline{N}^{\text{post}}$ in $\mathcal{L}^{0,h}$ (i.e. it reaches Λ_h). We then show that with a sufficiently large value of η (or φ), the increase in scent intensity going from $\overline{N}$ to $\overline{N}^{\text{post}}$ will more than compensate for the potential increase in boundary length. However, the transformation from $\overline{N}$ to $\overline{N}^{\text{post}}$ is complex, and we will thus split it into multiple steps – pre-process, the main transformation, and post-process. The layer sequences after each of these steps are denoted $\overline{N}^{\text{pre}}$, $\overline{N}^{\text{mid}}$ and $\overline{N}^{\text{post}}$ respectively.

To understand the following transformations from $\overline{N}$ to $\overline{N}^{\text{pre}}$ to $\overline{N}^{\text{mid}}$ to $\overline{N}^{\text{post}}$, we can visualize layer sequences as "right-justified" configurations in Λ (see Figure 3). The particles of these right-justified configurations can be grouped into columns, where the site on the k^{th} layer of column j (counted from the right) is filled if and only if its layer sequence has at least j particles in layer k.

The pre-processing step, which creates an intermediate layer sequence $\overline{N}^{\text{pre}} = (n_k^{\text{pre}})_{k\in[h]} \in \mathcal{L}^{0,h}$, is solely to account for specific edge cases in the eventual transformation to $\overline{N}^{\text{post}}$. To define $\overline{N}^{\text{pre}}$ starting from $\overline{N}$, we first add in the unused $n - |\sigma|$ particles to the layer sequence. We denote $u = w - \max_{k\in[h]}\{n_k\} - 1$, the number of empty columns that we can fill with unused particles. We thus take $\min\left\{u, \lfloor\frac{n-|\sigma|}{h}\rfloor\right\}$ empty columns and fill them with unused particles (which is equivalent to increasing each entry of the layer sequence by said amount of columns). The next step is to guarantee at least one completely filled column. If there are no completely filled columns even after adding the unused particles, $h - D$ particles are taken one at the time from the highest layer with at least two particles, and placed to form a single column of particles. This transformation is illustrated in Figure 3c.

The particles of the layer sequence $\overline{N}^{\text{pre}}$ can be divided into three layer sequences $(n_k^{\text{full}})_{k\in[h]}$, $(n_k^{\perp})_{k\in[h]}$ and $(n_k^{\top})_{k\in[h]}$ by column (so that $n_k^{\text{pre}} = n_k^{\text{full}} + n_k^{\perp} + n_k^{\top}$ for all $k \in [h]$).



(a) Original configuration σ with some number of unused particles (not shown).

(b) Representation of layer sequence $\overline{N}$ (right-justified σ) with unused particles not shown.

(c) Representation of layer sequence $\overline{N}^{\text{pre}}$. There were no empty columns in $\overline{N}$, so all unused particles remain unused, while 2 particles from the topmost layers are moved to form a single column to the top. Particles from $(n_k^{\text{full}})_{k\in[h]}$, $(n_k^{\perp})_{k\in[h]}$ and $(n_k^{\top})_{k\in[h]}$ represented by black circles, double circles and single circles respectively.

(d) Representation of layer sequence $\overline{N}^{\text{mid}}$ (excluding the crosses). Particles from $(n_k^{\top})_{k\in[h]}$ are shifted upward, and two full columns are created from the 26 particles in $(n_k^{\perp})_{k\in[h]}$, with 6 particles from $(n_k^{\perp})_{k\in[h]}$ unused. These 6 particles are added back to the positions marked with crosses to form the layer sequence $\overline{N}^{\text{post}}$.

Figure 3 Transformation from the layer sequence $\overline{N}$ to $\overline{N}^{\text{post}}$, with the layer sequences represented as right-justified configurations.

The first layer sequence $(n_k^{\text{full}})_{k\in[h]}$ is represents the particles belonging to the $J^{\text{full}} = n_h^{\text{pre}} \geq 1$ full columns of the right-justified configuration. The remaining columns are then identified as either *bottom-supported* or *non-bottom-supported*. The bottom-supported columns are those which have some value $k' \in \{0, 1, 2, \ldots, h-1\}$ where layers $\{1, 2, \ldots, k'\}$ in the column are occupied and the remaining layers are unoccupied (an alternative definition is having no unoccupied site below an occupied site). Particles from bottom-supported columns are put into $(n_k^{\perp})_{k\in[h]}$, while those from non-bottom-support columns go into $(n_k^{\top})_{k\in[h]}$. Alternatively, for each value of k in $[h]$ we can define formally

- $n_k^{\text{full}} = J^{\text{full}}$,
- $n_k^{\perp} = \left|\left\{j \in [w] : j > J^{\text{full}} \text{ and } n_{k'+1}^{\text{pre}} \geq j \implies n_{k'}^{\text{pre}} \geq j \text{ for all } k' \in [h-1]\right\}\right|$, and
- $n_k^{\top} = n_k^{\text{pre}} - n_k^{\perp} - n_k^{\text{full}}$.

To construct $\overline{N}^{\text{mid}}$, we apply different transformations to the particles corresponding to each of these layer sequences to construct a new configuration σ^{mid}. The particles in $(n_k^{\perp})_{k\in[h]}$ are transformed into $J^{\perp} = \lfloor \frac{1}{h} \sum_{k=1}^{h} n_k^{\perp} \rfloor$ full columns while the particles in $(n_k^{\top})_{k\in[h]}$ have their particles shifted to the tops of their respective columns. These non-bottom-supported columns are placed directly next to the $J^{\text{full}} + J^{\perp}$ filled columns in the right-justified configuration σ^{mid} (see Figure 3d). Note that there may be up to $h-1$ unused particles from $(n_k^{\perp})_{k\in[h]}$ after this transformation. The layer sequence of this new configuration will be denoted by $\overline{N}^{\text{mid}} = (n_k^{\text{mid}})_{k\in[h]}$.

To describe this transformation more precisely in terms of layer sequences, for each $i \in [w]$, we denote $\mathcal{C}_i^{\top} = |\{k \in [h] : n_k^{\top} \geq i\}|$, the number of particles in the i^{th} non-bottom-supported column. With J^{full} and $J^{\perp}$ defined as before, we can then define the layer sequence $\overline{N}^{\text{mid}} = (n_k^{\text{mid}})_{k\in[h]}$, where for each $k \in [h]$,

$$n_k^{\text{mid}} = J^{\text{full}} + J^{\perp} + \left|\left\{i \in [w] : \mathcal{C}_i^{\top} \geq h - k + 1\right\}\right|.$$

We note that this is a valid transformation (into a layer sequence $\overline{N}^{\text{mid}} \in \mathcal{L}^{0,h}$) as for each $k \in [h]$, we know that $J^{\perp} \leq \max_{k'}\{n_{k'}^{\perp}\}$ and $\left|\left\{i \in [w] : \mathcal{C}_i^{\top} \geq h - k + 1\right\}\right| \leq \max_{k'}\{n_{k'}^{\top}\}$, so we must have $n_k^{\text{mid}} \leq n_k^{\text{pre}} \leq w - 1$.

We include one final step in the transformation to ensure that the change in scent intensity from $\overline{N}^{\text{pre}}$ to $\overline{N}^{\text{post}}$ is non-negative. In particular, in the transformation from $\overline{N}^{\text{pre}}$ to $\overline{N}^{\text{post}}$, for the sake of simplicity, we had transformed the particles from $(n_k^{\perp})_{k\in[h]}$ into $J^{\perp} = \lfloor \frac{1}{h}\sum_{k=1}^{h} n_k^{\perp} \rfloor$ full columns, leaving $m \leq h-1$ particles unused. As a final post-processing step, we can add these m particles back to the final layer sequence, by adding one particle each to the topmost m rows that have less than $w-1$ particles (note that the final layer sequence is non-decreasing). This gives a positive net change in scent intensity for the particles from $(n_k^{\perp})_{k\in[h]}$, while potentially increasing the value of $\overline{B}_h$ by at most $O_h(1)$ (using the definition of $\overline{B}_h$ in Lemma 16).

Change in scent intensity. We compute the overall change in the two transformations to show Lemma 17. It is important to note that in our analysis, we do not assume that φ is constant with respect to h. Note that for this and the following lemmas, we will only give the brief ideas for the proofs; the remaining details are presented in [17]. Recall that $\overline{S}(\overline{N}')$ denotes the scent intensity common to all configurations with layer sequence $\overline{N}'$.

▶ **Lemma 17** (Scent Intensity). *The changes in scent intensities in the transformations from $\overline{N}$ to $\overline{N}^{pre}$ and from $\overline{N}^{pre}$ to $\overline{N}^{post}$ are non-negative and have the following lower bounds:*

$$\overline{S}(\overline{N}^{pre}) - \overline{S}(\overline{N}) \geq \varphi \cdot \min\left\{u, \left\lfloor \frac{n - |\sigma|}{h} \right\rfloor\right\}$$

$$\overline{S}(\overline{N}^{post}) - \overline{S}(\overline{N}^{pre}) \geq |\sigma|\frac{\varphi}{h}\left(1 - \frac{D}{h}\right) + \varphi \cdot O_h(1),$$

where $u = w - \max_{k\in[h]}\{n_k\} - 1$ and $O_h(1)$ represents a function that does not depend on the specific choice of starting configuration σ.

For the transformation from $\overline{N}$ to $\overline{N}^{\text{pre}}$ in the proof of Lemma 17, the number of full columns added to the $\overline{N}$ is $\min\left\{u, \left\lfloor \frac{n-|\sigma|}{h} \right\rfloor\right\}$, and neither the adding of full columns nor the shifting of $h - D$ particles to the top $h - D$ rows can decrease the total scent intensity. On the other hand, the transformation from $\overline{N}^{\text{pre}}$ to $\overline{N}^{\text{post}}$ requires a more involved analysis, to show that for particles from both bottom-supported columns and non-bottom-supported columns, the average increase in the total scent intensity per particle has a lower bound of $\frac{\varphi}{h}\left(1 - \frac{D}{h}\right)$, with a small error term.

Change in boundary length. Instead of computing the boundary lengths for configurations directly, we compare the (tight) boundary length lower bounds that are defined in Lemma 16 between the initial and final layer sequences $\overline{N}$ and $\overline{N}^{\text{post}}$.

▶ **Lemma 18** (Boundary Length). *In the transformation from $\overline{N} \in \mathcal{L}^{0,D}$ to $\overline{N}^{post} \in \mathcal{L}^{0,h}$, we have:*

$$\overline{B}_h(\overline{N}^{post}) - \overline{B}_D(\overline{N}) \leq 4(h - D) - 2\max\left\{w - u - D, \left(\frac{|\sigma|}{h} - \frac{h}{2}\right)\right\} + O_h(1),$$

where $O_h(1)$ represents a function that does not depend on the specific choice of starting configuration σ. Furthermore, by Lemma 16 there exists a configuration τ with layer sequence $\overline{N}^{post}$ such that $B(\tau) - B(\sigma) \leq \overline{B}_h(\overline{N}^{post}) - \overline{B}_D(\overline{N})$, and thus has the same upper bound.

The proof of Lemma 18 focuses almost entirely on the comparison between layer sequences $\overline{N}$ and $\overline{N}^{\mathrm{mid}}$, by noting that the post-processing step produces an insignificant change in boundary length.

Change in the Hamiltonian. Finally, putting together the changes in scent intensity (Lemma 17) and boundary length (Lemma 18) going form $\overline{N}$ to $\overline{N}^{\mathrm{post}}$, we can compute a lower bound for the change in Hamiltonian. There are two cases in the proof of Lemma 13, roughly corresponding to whether or not there were enough empty columns u in $\overline{N}$ to fit the majority of the unused particles in the transformation from $\overline{N}$ to $\overline{N}^{\mathrm{pre}}$. In the case where u is small ($u < \lfloor \frac{n-|\sigma|}{h} \rfloor$), the boundary length of the initial configuration or layer sequence will be close to w, and is the primary contributor to the change in Hamiltonian. In the case where u is large ($u \geq \lfloor \frac{n-|\sigma|}{h} \rfloor$), the increase in scent intensity without additional boundary length from being able to construct full columns of particles makes it undesirable for particles to remain unused in the configuration.

Memory Lower Bounds and Impossibility Results for Anonymous Dynamic Broadcast

Garrett Parzych

School of Computing and Augmented Intelligence, Biodesign Center for Biocomputing, Security and Society Arizona State University, Tempe, AZ, USA

Joshua J. Daymude

School of Computing and Augmented Intelligence, Biodesign Center for Biocomputing, Security and Society Arizona State University, Tempe, AZ, USA

Abstract

Broadcast is a ubiquitous distributed computing problem that underpins many other system tasks. In static, connected networks, it was recently shown that broadcast is solvable without any node memory and only constant-size messages in worst-case asymptotically optimal time (Hussak and Trehan, PODC'19/STACS'20/DC'23). In the dynamic setting of adversarial topology changes, however, existing algorithms rely on identifiers, port labels, or polynomial memory to solve broadcast and compute functions over node inputs. We investigate *space-efficient, terminating broadcast algorithms for anonymous, synchronous, 1-interval connected dynamic networks* and introduce the first memory lower bounds in this setting. Specifically, we prove that broadcast with *termination detection* is impossible for idle-start algorithms (where only the broadcaster can initially send messages) and otherwise requires $\Omega(\log n)$ memory per node, where n is the number of nodes in the network. Even if the termination condition is relaxed to *stabilizing termination* (eventually no additional messages are sent), we show that any idle-start algorithm must use $\omega(1)$ memory per node, separating the static and dynamic settings for anonymous broadcast. This lower bound is not far from optimal, as we present an algorithm that solves broadcast with stabilizing termination using $\mathcal{O}(\log n)$ memory per node in worst-case asymptotically optimal time. In sum, these results reveal the necessity of non-constant memory for nontrivial terminating computation in anonymous dynamic networks.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Networks → Network algorithms

Keywords and phrases Dynamic networks, anonymity, broadcast, space complexity, lower bounds, termination detection, stabilizing termination

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.35

Funding This work is supported in part by National Science Foundation award CCF-2312537.

1 Introduction

Distributed algorithms for *dynamic networks* enable processes to coordinate even as the communication links between them change over time, often rapidly and adversarially [1,8]. When once a process disconnecting from a distributed system was viewed as a rare crash fault to tolerate, research over the last two decades has come to view dynamics as natural or even necessary to a system's function. Application domains such as self-stabilizing overlay networks [2,19], blockchains [4,21], and swarm robotics [20,22] are defined by their rapidly changing or physically moving components, forcing algorithms to achieve their goals by leveraging – or more often operating in spite of – these dynamics.

Despite the challenges, many fundamental problems have been addressed under adversarial dynamics, including broadcast, consensus, and leader election (see [3,5] for complementary surveys). However, many of these algorithms endow their nodes with unique identifiers, port labels for locally distinguishing among their neighbors, (approximate) knowledge of

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 35; pp. 35:1–35:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the number of nodes in the network, or superlinear memories. Taking inspiration from collective behavior in biological complex systems of computationally weak entities – such as foraging in ant colonies [10, 32], aggregation in slime mold spore migrations [36], and energy distribution in microbiomes [28, 33] – we question to what extent these additional capabilities are necessary. Specifically, we consider dynamic networks of *anonymous* nodes (lacking both unique identifiers and port labels) with limited memory. As an aside, anonymity is a desirable feature in its own right for engineering privacy-sensitive applications, such as Bluetooth-based contact tracing [35].

We consider the fundamental problem of synchronous *broadcast*, in which all nodes in a network must eventually be informed of some information originating at a single node. In static, connected networks, broadcast can be solved without any persistent memory at all: the *amnesiac flooding* algorithm [23–25] – in which nodes forward copies of any message they receive to any neighbor that did not send them the message in the last round – informs all nodes (correctness) and allows them to eventually stop sending additional messages (stabilizing termination) within worst-case asymptotically optimal time. But even when adversarial dynamics are constrained to maintain network connectivity in every round, a longstanding conjecture states that dynamic broadcast with stabilizing termination is impossible in $\mathcal{O}(\log n)$ memory without node identifiers or knowledge of n [31], let alone without any memory at all. We summarize our contributions as follows.

Our Contributions. All results are proven with respect to deterministic algorithms run by anonymous nodes (lacking identifiers and port labels) in a dynamic network whose topology can change arbitrarily but remains connected in each synchronous round. In this setting:

- Broadcast with *termination detection* – i.e., the broadcaster must eventually decide broadcast is complete – is impossible for idle-start algorithms where only the broadcaster can initially send messages (Section 2) and otherwise requires $\Omega(\log n)$ space (Section 3).
- Any idle-start algorithm solving broadcast with *stabilizing termination* – i.e., eventually no additional messages are sent – must have $\omega(1)$ space complexity (Section 4). We then present an algorithm solving broadcast with stabilizing termination in $\mathcal{O}(\log n)$ space and worst-case asymptotically optimal time (Section 5).

We note that although synchronous systems are typically seen as less general than asynchronous ones, it is actually the opposite for the purposes of lower bounds. Since an asynchronous adversary can always simulate a synchronous one, any impossibility results or lower bounds proven w.r.t. synchrony will apply to both types of systems.

1.1 Model

Dynamic Networks. We consider a synchronous dynamic network comprising a fixed set of nodes V. Nodes communicate with each other via message passing over a communication graph whose topology changes over time. We model this topology as a *time-varying graph* $\mathcal{G} = (V, E, T, \rho)$ where V is the set of nodes, E is the static set of undirected edges that may appear in the graph, $T = \mathbb{N}$ is the lifetime of the graph, and $\rho : E \times T \to \{0, 1\}$ is the presence function indicating whether an edge exists at a given time [8]. We refer to the set of edges present at time $t \in T$ as $E_t = \{e \in E : \rho(e, t) = 1\}$ and the undirected graph $G_t = (V, E_t)$ as the *snapshot* of $\mathcal{G}$ at time $t \in T$. We assume an adversary controls the presence function ρ and that E is the complete set of edges on V; i.e., we do not limit which edges the adversary can introduce. We do, however, follow the majority of dynamic broadcast literature (e.g., [8, 12, 26, 27, 31]) in assuming *1-interval-connectivity* (also called "always-connected snapshots"); i.e., the adversary may make arbitrary topological changes at each time $t \in T$ so long as each snapshot G_t is connected.

Node Capabilities. Motivated by computationally weak individuals in biological collectives (e.g., cells, microbes, social insects, etc.), we consider nodes that are *anonymous*, lacking unique identifiers, and have *no knowledge or approximation of any global measure*, including the number of nodes n. We further assume that nodes have *no port labels*; i.e., they cannot count or locally distinguish among their neighbors. Consequently, when a node communicates with its neighbors via *message passing*, it does so using a broadcast mechanism, sending the same message to all its current neighbors.

Algorithms and Execution. Each node in the time-varying graph $\mathcal{G}$ synchronously executes the same distributed algorithm $\mathcal{A}$. All nodes are initialized at time $t = 0$, and each synchronous round t starting at time t proceeds as follows:

1. The adversary fixes the network topology G_t for round t.
2. Each node may send a message to its neighbors in G_t according to algorithm $\mathcal{A}$ as a function of its current state.
3. Each node may perform a state transition according to algorithm $\mathcal{A}$ as a function of its current state and the multiset of messages it (reliably) receives from its neighbors in G_t.

Memory. In this paper, we are primarily concerned with an algorithm's *space complexity*, the maximum number of bits a node uses to store its state between rounds. As usual for distributed systems, we are interested in the asymptotic growth of this measure as a function of $n = |V|$, the number of nodes. We emphasize that even if nodes have $\Omega(\log n)$ memory – sufficient for storing unique identifiers – they are anonymous and are not assigned such identifiers a priori. We also note that we do not analyze *message complexity* directly, as our execution model specifies nodes that send messages based only on their states; thus, there are at most as many message types as states.

Broadcast. In the *broadcast* problem, every node starts in the same state except for a single node known as the *broadcaster* that is trying to deliver some information to every other node in the network. We say that a node is *informed* if it is the broadcaster or has previously received a message from an informed node. All other nodes are *uninformed.* A broadcast is *complete* when every node in the network is informed. A distributed algorithm $\mathcal{A}$ *solves* the broadcast problem in t rounds if, for any time-varying graph whose nodes all execute $\mathcal{A}$, broadcast is completed by the end of round t.

Idle-Start. A node is *idle* if it will not send a message in the subsequent round, and will not change its state if it does not receive a message (this has also been called "quiescent" [29], though that term is overloaded in this context). Some broadcast algorithms critically rely on initializing all nodes as non-idle at time $t = 0$; conversely, an *idle-start* algorithm initializes all nodes except the broadcaster as idle.

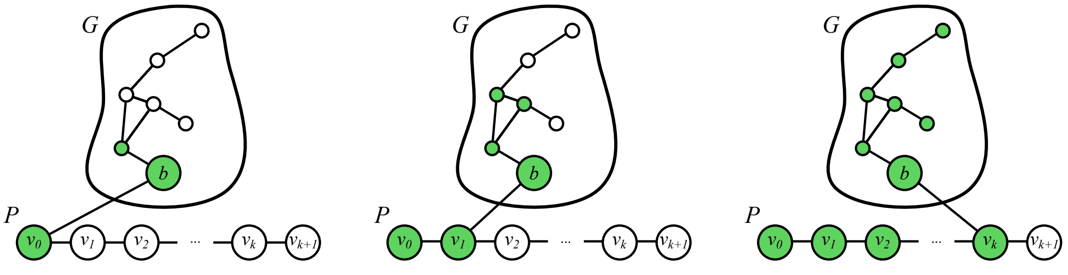
Termination. A simple solution to broadcast is to make every informed node continuously send messages. Since we assume 1-interval connectivity, there is always at least one uninformed node receiving a message from an informed node in each round, so this algorithm solves broadcast in $\mathcal{O}(n)$ rounds. This runtime bound is worst-case asymptotically optimal, but a smoothed analysis reveals significant improvements on more "typical" topologies [17, 18]. This algorithm also achieves $\Theta(1)$ space complexity, since nodes need only remember whether they're informed. However, nodes sending messages forever creates undesirable congestion and precludes the system from advancing to further tasks, e.g., starting a new broadcast or using the broadcast information as part of a larger algorithm. Thus, we seek algorithms meeting some kind of termination conditions. An algorithm achieves *stabilizing termination* if every node becomes idle within finite time. An algorithm achieves the stronger condition of *termination detection* if the broadcaster correctly and irrevocably decides that broadcast is complete (i.e., by entering a terminating state) within finite time.

1.2 Related Work

Broadcast is a ubiquitous and well-studied distributed computing problem, often appearing as a building block in more complex tasks. In static, connected networks, broadcast with stabilizing termination is solvable without any node memory (and thus without identifiers) and only $\Theta(1)$ message complexity in worst-case asymptotically optimal time, though port labels are required to distinguish among neighbors [23,24]. In a recent extension of this work, the same algorithm was proven correct under node and edge deletion dynamics [25], but breaks down under more general adversarial dynamics. With this inspiration, our focus is space-efficient, terminating algorithms for broadcast in anonymous dynamic networks.

Early works on dynamic broadcast typically assumed stronger node capabilities. A series of works on shortest, fastest, and foremost broadcast assumed local identifiers enabling a node u to maintain a consistent label for any neighbor v, even if v disconnected from and later reconnected to u [6–8]. This assumption enables the construction of time-invariant logical structures like spanning trees, which – when combined with the assumption of recurrent dynamics (edges will eventually reappear) – reduces dynamic broadcast to static routing on these structures. Similar techniques are used when assuming both unique node identifiers and shared knowledge of n, the number of nodes in the network [34]. Among these early works, only O'Dell and Wattenhofer [31] share our focus on anonymous nodes and space complexity. They conjectured that no algorithm can solve broadcast with stabilizing termination in $\mathcal{O}(\log n)$ space when nodes are anonymous and have no knowledge of n. Interestingly, we prove this is solvable in our model (Section 5), though we do not disprove their conjecture since our model's synchrony is not directly comparable to their combination of asynchronous time, bounded message latency, and disconnection detection for re-broadcasting messages.

A parallel line of work investigated what functions a dynamic network can deterministically compute over its nodes' inputs [27]. In the context of anonymous dynamic networks, most results focus on the *(exact) counting problem* [9,11,14,15,26,30], which terminating broadcast reduces to in $\mathcal{O}(n)$ time and $\mathcal{O}(\log n)$ space: once the broadcaster knows the number of nodes n, it need only wait n rounds before every other node must have been informed (since the dynamic network is 1-interval connected), at which point it can terminate. These works recently culminated in the exact characterization by Di Luna and Viglietta [12,13] showing that anonymous dynamic networks with at least one leader can compute only the *multi-aggregate functions* – those for which a node's output depends only on its own input and the multiset of all nodes' inputs – and do so in optimal (linear) time. However, their algorithm uses $\Theta(n^3 \log n)$ space in the worst case [16], leaving open what memory is necessary. Our impossibility results and memory lower bounds for terminating broadcast (Sections 2–4) shed light on this question, as many nontrivial multi-aggregate functions require information from at least one node to be communicated to all other nodes (e.g., minimums/maximums, averages, exact and generalized counting, etc.).



(a) End of Round 0. **(b)** End of Round 1. **(c)** End of Round k.

Figure 1 The time-varying graph $\mathcal{G}$ used in the proof of Theorem 1. In each round t, the broadcaster b is connected to node v_t in the path P. Informed nodes are shown in green. When b declares broadcast to be complete in round k, node v_{k+1} is still uninformed.

2 Impossibility Results for Termination Detection

We begin by proving that there is no idle-start algorithm – i.e., one in which only the broadcaster can initially send messages – that solves broadcast with termination detection in our setting. The idea behind the proof is as follows. Supposing to the contrary that such an algorithm $\mathcal{A}$ exists, it must solve broadcast and detect termination on any static, connected network G. So we consider an execution of $\mathcal{A}$ on an extension of G as a time-varying graph $\mathcal{G}$ that we carefully construct to achieve two goals: (1) the execution of $\mathcal{A}$ on G in the time-varying graph $\mathcal{G}$ is identical to its execution on G alone, and (2) there is an idle node in $\mathcal{G}$ that is sequestered from ever being informed. This drives a contradiction: the broadcaster must detect termination in $\mathcal{G}$ because it does so on G alone, but will do so incorrectly because there is still an uninformed node.

▶ **Theorem 1.** *No deterministic idle-start algorithm can solve broadcast with termination detection for anonymous, synchronous, 1-interval connected dynamic networks.*

Proof. Suppose to the contrary that there exists an idle-start algorithm $\mathcal{A}$ solving broadcast with termination detection in our setting. Let G be any static, connected graph and let b be any node in G. If $\mathcal{A}$ is executed on G with b as the broadcaster, there must exist some round k during which b correctly and irrevocably declares broadcast to be complete.

Construct a time-varying graph $\mathcal{G}$ as follows (see Figure 1). First, $\mathcal{G}$ contains the static graph G as a fixed part; i.e., all edges of G will remain present throughout the lifetime of $\mathcal{G}$. Additionally, $\mathcal{G}$ contains a fixed path $P = v_0v_1 \cdots v_{k+1}$ of $k+2$ nodes. Finally, in each round $t \in \{0, \ldots, k\}$, there is a single edge $\{b, v_t\}$ connecting the broadcaster b to the path P.

Consider the execution of $\mathcal{A}$ on $\mathcal{G}$ with b as the broadcaster. We argue by induction on $t \in \{0, \ldots, k\}$ that in round t, (1) all nodes in G send the same messages and perform the same state transitions as they did in the execution of $\mathcal{A}$ on G, and (2) the nodes $\{v_{t+1}, \ldots, v_{k+1}\}$ remain idle and uninformed. In round $t = 0$, only the broadcaster b potentially sends messages and changes state since $\mathcal{A}$ is an idle-start algorithm. Recall that nodes have no port labels and no knowledge of n. Thus, b must send the same messages and perform the same state transition as in the execution of $\mathcal{A}$ on G. The only neighbor of b in the path P is v_0, so nodes $\{v_1, \ldots, v_{k+1}\}$ receive no messages and remain idle and uninformed.

Now suppose the claim holds up to and including some round $0 \leq t < k$. By the induction hypothesis, all nodes in G have the same states at the start of round $t+1$ as in the execution of $\mathcal{A}$ on G, and thus send the same messages. However, since $\{b, v_{t+1}\}$ is the only edge between G and P in round $t+1$, b could in principle make a divergent state transition if it receives a message from v_{t+1}. But by the induction hypothesis, v_{t+1} is idle and thus sends no messages in round $t+1$. For this same reason, nodes $\{v_{t+2}, \ldots, v_{k+1}\}$ receive no messages and remain idle and uninformed in round $t+1$.

Therefore, in round k, the broadcaster b irrevocably declares broadcast to be complete just as it did in the static setting, but v_{k+1} remains uninformed, a contradiction. ◀

A related line of research investigates the computability of functions in dynamic networks with no or multiple leaders. Di Luna and Viglietta have recently shown that if the number of leaders is known, nodes can compute the same functions as systems with a single leader. However, they found that if the number of leaders is unknown, then it is impossible for nodes to compute the size of the network with termination detection [13]. Using similar ideas to our previous proof, we can show that broadcast with termination detection is also impossible without knowing the number of broadcasters. Since broadcast can be reduced to counting the size of the network (simply broadcast for n rounds after receiving the count), our theorem implies the result of Di Luna and Viglietta. However, since we are unaware of any reduction from counting to broadcast, ours seems to be slightly more general.

▶ **Theorem 2.** *Even without an idle start, no deterministic algorithm can solve broadcast with termination detection for anonymous, synchronous, 1-interval connected dynamic networks if nodes have no knowledge of the number of broadcasters.*

Proof. Suppose for contradiction that an algorithm $\mathcal{A}$ solves broadcast with termination detection without giving nodes knowledge of the number of broadcasters. Let the *configuration* of a dynamic network at time t be the multiset of node states at the start of round t. Let $C_0, C_1, \ldots, C_x$ be the sequence of configurations that occur from running $\mathcal{A}$ on the complete graph K_3 with a single broadcaster, where C_x is the first configuration in which the broadcaster declares termination. We will create a new time-varying graph $\mathcal{G}$ on which $\mathcal{A}$ will incorrectly terminate. First create 2^x copies of C_0, with each copy having a single broadcaster and two non-broadcasters. Add a path $p_0p_1 \ldots p_x$ of $x+1$ nodes and for each copy of C_0, choose one of the non-broadcaster nodes and attach it to p_0; this will be the first snapshot of $\mathcal{G}$.

Suppose that in C_1, the broadcaster was in state β_1 and the non-broadcasters in state α_1. By symmetry of the graphs, after the first round of executing $\mathcal{A}$ on $\mathcal{G}$, each copy of C_0 will have a node in state β_1 and a node in state α_1, while the node attached to p_0 will be in some other unknown state. Thus there are 2^x nodes in state β_1 and 2^x nodes in state α_1. Use all the nodes in state α_1 and half the nodes in state β_1 to create 2^{x-1} copies of the configuration C_1. Attach all of the unused nodes from the copies of C_0 to p_1. Again, choose a single non-broadcaster node from each copy of C_1, and attach them to p_1. Then run $\mathcal{A}$ for one additional round. If β_2 and α_2 are the states of the broadcaster and non-broadcasters in C_2 respectively, then by symmetry, our graph after this round will have 2^{x-1} copies each of β_2 and α_2. We can now use these nodes to create 2^{x-2} copies of C_2.

Continuing on in this way, we can create 2^{x-i} copies of configuration C_i for each $i \in \{0, \ldots, x\}$ while only informing a single node in the path at a time. Thus, after x rounds, we will have $2^{x-x} = 1$ copy of C_x and p_x will still be uninformed. But the broadcaster in the copy of C_x will have declared termination, contradicting the correctness of $\mathcal{A}$. ◀

3 Memory Lower Bound for Termination Detection

In an idle-start algorithm, nodes have no indication of whether they have idle, uninformed neighbors. This drives the indistinguishability result at the center of Theorem 1: a static, connected graph cannot tell if it's the entire network or a subgraph in a larger whole. Without

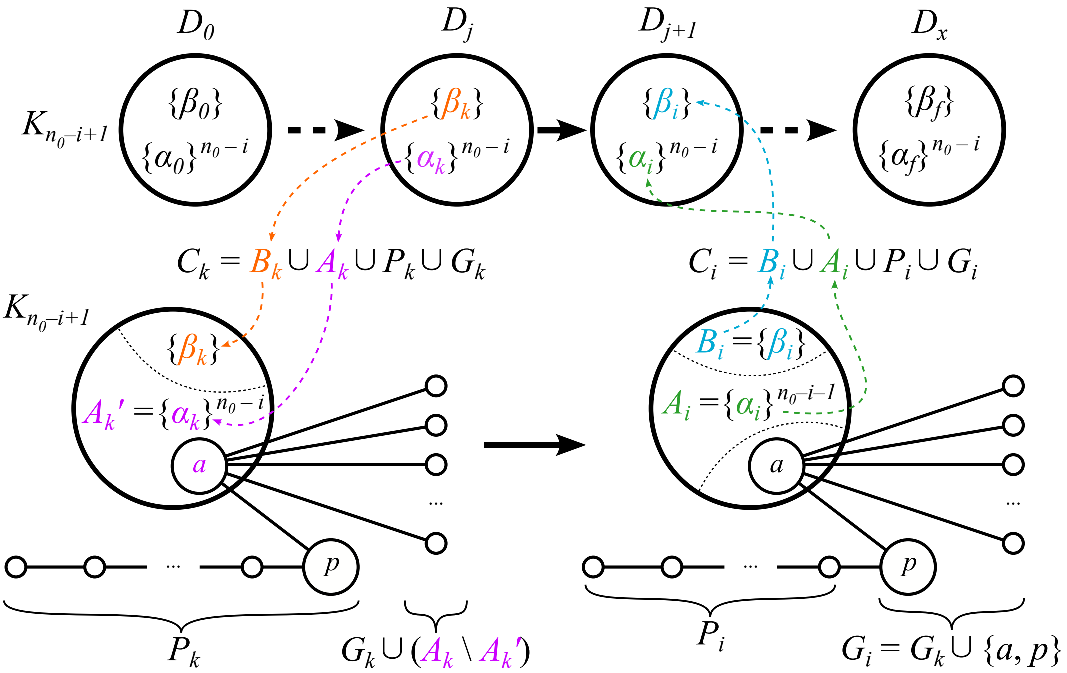


Figure 2 The inductive construction of configuration C_i as described in the proof of Theorem 3. The key idea is to find a new pair of states (β_i, α_i) that is not already in $(\beta_0, \alpha_0), \ldots, (\beta_{i-1}, \alpha_{i-1})$ by arranging a previously identified reachable configuration C_k and extending the corresponding execution of $\mathcal{A}$ by one additional round.

the constraints of an idle-start, however, this particular contradiction – and its corresponding impossibility result – disappears. For example, the history tree algorithm of Di Luna and Viglietta solves broadcast in this setting in linear time and $\Theta(n^3 \log n)$ space [16]. Still, all such algorithms must use at least logarithmic memory, as we now show.

▶ **Theorem 3.** *Any algorithm that solves broadcast with termination detection for anonymous, synchronous, 1-interval connected dynamic networks must have* $\Omega(\log n)$ *space complexity.*

Proof. Consider any (non-idle-start) algorithm $\mathcal{A}$ that solves broadcast with termination detection. Let $f(n)$ be the maximum number of states that $\mathcal{A}$ uses when run on dynamic networks of at most n nodes. We will show that $f(n) \geq n^{1/2}$ for all $n \geq 1$, implying that $\mathcal{A}$ uses $\log(f(n)) \geq \log(n^{1/2}) = \Omega(\log n)$ space.

Suppose to the contrary that there exists an $n_0 \geq 1$ such that $f(n_0) < n_0^{1/2}$. A configuration C is *reachable* (from an initial configuration) if there exists a time-varying graph $\mathcal{G}$ and time t such that the execution of $\mathcal{A}$ on $\mathcal{G}$ for t rounds results in configuration C. We will find a sequence of n_0 reachable configurations $(C_i)_{i=0}^{n_0-1}$ where each C_i is a disjoint union of multisets[1] of states $B_i \cup A_i \cup P_i \cup G_i$ satisfying:

1. $|C_i| = 2n_0$,
2. $B_i = \{\beta_i\}$, where β_i is the state of the broadcaster,
3. $A_i = \{\alpha_i\}^{n_0-i-1}$, exactly $n_0 - i - 1$ copies of the same state α_i,

[1] A multiset X is a disjoint union of multisets $Y \cup Z$ if the multiplicity of any element $x \in X$ is the sum of multiplicities of x in Y and Z.

4. $|P_i| \geq n_0 - i$ and contains only uninformed states,
5. $G_i = C_i \setminus (B_i \cup A_i \cup P_i)$, and
6. $(\beta_i, \alpha_i) \neq (\beta_j, \alpha_j)$ for all $j \neq i$.

Initially, the broadcaster is in some state β_0 and all other nodes are uninformed in some state $\alpha_0 \neq \beta_0$. Define the initial configuration C_0 by letting $B_0 = \{\beta_0\}$, $A_0 = \{\alpha_0\}^{n_0-1}$, $P_0 = \{\alpha_0\}^{n_0}$, and $G_0 = \varnothing$. Clearly, C_0 is reachable and meets the above conditions.

Now consider any $1 \leq i < n_0$ and suppose that $C_0, \ldots, C_{i-1}$ have already been defined; we define C_i recursively as follows (see Figure 2 for an illustration). Let $\mathcal{D} = (D_0, \ldots, D_x)$ be the sequence of configurations obtained by running $\mathcal{A}$ on K_{n_0-i+1}, the complete graph on $n_0 - i + 1$ nodes, where D_x is the first configuration in which the broadcaster declares termination. Note that, by the symmetry of the complete graph, each configuration in $\mathcal{D}$ has one state for the broadcaster and one state shared by every other node. Let $j \in \{0, \ldots, x\}$ and $k \in \{0, \ldots, i-1\}$ be such that $\beta_k \in D_j$ and $D_j \setminus \{\beta_k\} \subseteq A_k$ and for any other j', k' fulfilling this condition, $j' \leq j$. Note that such a j must exist since, firstly, D_0 is the initial configuration of K_{n_0-i+1} and thus $\beta_0 \in D_0$ and $D_0 \setminus \{\beta_0\} = \{\alpha_0\}^{n_0-i} \subseteq \{\alpha_0\}^{n_0-1} = A_0$; and secondly, the sequence $\mathcal{D}$ is finite since $\mathcal{A}$ terminates in finite time. We also have $j < x$ since otherwise there exists a configuration C_k containing $|P_k| \geq n_0 - k > 0$ uninformed nodes (by induction), but $\beta_k \in D_x$ is a terminating state, contradicting the correctness of $\mathcal{A}$.

Configuration C_k is reachable, so there is some time-varying graph upon which the execution of $\mathcal{A}$ will within finite time be in configuration $C_k = B_k \cup A_k \cup P_k \cup G_k$. We extend this execution by putting this graph into the following topology and executing $\mathcal{A}$ for a single additional round. In a slight abuse of notation, we refer here to configurations as sets of nodes instead of multisets of states since edge dynamics make nodes in the same state interchangeable. Choose any subset $A'_k \subseteq A_k$ of $n_0 - i$ nodes (which is well-defined since $|A_k| = n_0 - k - 1 \geq n_0 - i$ by induction) and arrange them with B_k as a complete graph K_{n_0-i+1}. Arrange the nodes in P_k as a path. Connect these components by attaching an end $p \in P_k$ of the path to some node $a \in A'_k$ and connect every node in $G_k \cup (A_k \setminus A'_k)$ only to this node a. Let C_i be the configuration obtained after executing one round of $\mathcal{A}$ on this topology. Clearly C_i is reachable; we show next that it satisfies the required conditions.

1. $|C_i| = |C_k| = 2n_0$ as desired.
2. Let $B_i = \{\beta_i\}$ be the state of the unique node in B_k after this one additional round.
3. Consider any two nodes $a_1, a_2 \in A'_k \setminus \{a\}$ before execution of the round. Notice that they are both connected to B_k and every node in A'_k except for themselves. By induction, every node in A'_k has the same state α_k. Thus, a_1 and a_2 receive the same sets of messages from their neighbors, so they transition to the same state in the next round. Let α_i be this state shared by all nodes in $A'_k \setminus \{a\}$ after the round's execution and let A_i be the corresponding configuration. Thus $A_i = \{\alpha_i\}^{|A'_k|-1} = \{\alpha_i\}^{n_0-i-1}$ as desired.
4. Consider any node $v \in P_k \setminus \{p\}$ before execution of the round. By induction, v and its neighbors are uninformed. Thus, v receives no message from an informed node in the subsequent round and remains uninformed. Let P_i be the (states of) nodes $P_k \setminus \{p\}$. By induction, we have $|P_i| = |P_k| - 1 \geq n_0 - k - 1 \geq n_0 - i$ as desired.
5. $G_i = C_i \setminus (B_i \cup A_i \cup P_i)$ is simply defined as all the remaining nodes from C_k.
6. Suppose for contradiction that $(\beta_i, \alpha_i) = (\beta_\ell, \alpha_\ell)$ for some $\ell \in \{0, \ldots, i-1\}$. Recall that the configuration $D_j = \{\beta_k\} \cup \{\alpha_k\}^{n_0-i}$ was obtained by running $\mathcal{A}$ on K_{n_0-i+1} and D_j is not the final configuration of that execution. Moreover, observe that when we extended the execution reaching C_k by one round, we reconstructed this exact configuration and topology by arranging $A'_k \cup B_k$ as a complete graph. When running $\mathcal{A}$ for one

additional round on this component, the broadcaster in D_j and B_k transitions from β_k to β_i and the non-broadcasters in D_j and $A'_k \setminus \{a\}$ transition from α_k to α_i. Thus, $D_{j+1} = \{\beta_i\} \cup \{\alpha_i\}^{n_0-i} = \{\beta_\ell\} \cup \{\alpha_\ell\}^{n_0-i}$, by supposition. But this implies $\beta_\ell \in D_{j+1}$ and $D_{j+1} \setminus \{\beta_\ell\} = \{\alpha_\ell\}^{n_0-i} \subseteq \{\alpha_\ell\}^{n_0-\ell} = A_\ell$, contradicting the maximality of j.

Thus, we obtain the desired sequence of n_0 reachable configurations $(C_i)_{i=0}^{n_0-1}$ and their corresponding distinct state pairs (β_i, α_i). The initial state pair (β_0, α_0) appears in every execution of $\mathcal{A}$ and the remaining state pairs (β_i, α_i) for $i \geq 1$ appear in executions of $\mathcal{A}$ on complete graphs K_{n_0-i+1} of at most n_0 nodes. But the maximum number of states $\mathcal{A}$ can use on dynamic networks of at most n_0 nodes is $f(n_0) < n_0^{1/2}$, so there are only $f(n_0)^2 < n_0$ distinct state pairs, a contradiction. ◀

4 Memory Lower Bound for Stabilizing Termination from Idle-Start

We now turn our attention from termination detection to stabilizing termination, requiring only that all nodes are eventually informed and stop sending messages. In this section, we prove that any idle-start algorithm solving broadcast with stabilizing termination must use superconstant memory. This shows that stabilizing broadcast is strictly harder in dynamic networks than in the static setting, which has a trivial constant-memory algorithm (if uninformed and receiving a message, become informed and forward the message to all neighbors) and even permits an algorithm with no persistent memory at all [23, 24].

Our proof will use similar time-varying graphs as in the proofs of Theorems 1 and 3. However, those proofs derived contradictions from the broadcaster declaring termination too early, a condition that cannot be used in the case of stabilizing termination. Instead, we suppose a constant memory algorithm exists and use it to create an infinite sequence of configurations satisfying some very specific properties. We show in Lemma 6, however, that no infinite sequence with these properties exists.

In Lemma 5, we show that the time an algorithm takes to stabilize on a static complete graph, starting from any reachable configuration, depends only on the number of states used and not on the number of nodes in the graph. This will be useful in our proof since we consider algorithms using constant memory, thus implying a fixed bound on the stabilization time for any complete graph, regardless of size. To this end, we define the following function.

▶ **Definition 4.** *Let $f(k)$ be the minimum number of rounds such that for any idle-start algorithm $\mathcal{A}$ solving broadcast with stabilizing termination using at most k states and any configuration C reachable by $\mathcal{A}$, an execution of $\mathcal{A}$ on a static complete graph starting in configuration C stabilizes within $f(k)$ rounds.*

We first prove that this function is well-defined and bounded.

▶ **Lemma 5.** $f(k) \leq 3k!$

Proof. Let $\mathcal{A}$ be an idle-start algorithm solving broadcast with stabilizing termination using only k states and consider its execution on a static complete graph starting from some reachable configuration C. At each round, consider partitioning the nodes into $\ell \leq k$ sets by their current state. In each round, any nodes with the same state receive the same messages from their neighbors and transition to the same next state, so nodes sharing a state continue to do so throughout the execution. Thus, although the number of sets ℓ can decrease over time as some nodes converge to the same state, it can never increase. If there are ℓ sets of nodes with distinct states at one time, there are $\binom{k}{\ell}$ possibilities for what these states are and $\ell!$ ways for these states to be assigned to the sets. Again, since the number of of sets can decrease, ℓ may take on any value in $\{1, \ldots, k\}$ throughout the execution.

Since $\mathcal{A}$ is deterministic, if the same assignment of states to the same sets ever occurs twice in an execution, the algorithm must be in a loop. However, this can not happen since C is reachable and thus $\mathcal{A}$ must stabilize in finite time. Thus, the maximum number of configurations $\mathcal{A}$ can visit before stabilizing is

$$\sum_{\ell=1}^{k} \binom{k}{\ell} \cdot \ell! = \sum_{\ell=1}^{k} \frac{k!}{(k-\ell)!} = k! \cdot \sum_{i=0}^{k-1} \frac{1}{i!} \leq k! \cdot e \qquad \blacktriangleleft$$

Our next lemma sets us up for the contradiction in Theorem 7.

▶ **Lemma 6.** *Let $\mathcal{S} = \{S_1, S_2, \dots\}$ be a collection of multisets S_i which are each the disjoint union of multisets $A_i \cup B_i$ with elements from $[n] = \{1, \dots, n\}$ satisfying $|A_i| = k$, $|B_i|$ is finite, and $\neg(A_j = A_i \wedge B_j \subseteq B_i)$ for all $i \neq j$. Then $|\mathcal{S}|$ is finite.*

Proof. Suppose for contradiction that $\mathcal{S}$ is an infinite collection fulfilling these conditions. Since each A_i contains elements from $[n]$ and $|A_i| = k$, there are only a finite number of possible definitions for the multiset A_i. Since $\mathcal{S}$ is infinite, it must contain some infinite subcollection $\mathcal{S}' = \{S'_1, S'_2, \dots\} \subseteq \mathcal{S}$ such that $A'_i = A'_j$ for all $S'_i, S'_j \in \mathcal{S}'$. Thus, for $\mathcal{S}$ to fulfill the conditions, we must have $B'_i \not\subseteq B'_j$ for all $S'_i \neq S'_j$. We will derive a contradiction by finding multisets in $\mathcal{X}_0 = \{B'_i \mid S'_i \in \mathcal{S}'\}$ such that one is a subset of the other. In fact, we show something much more general: $\mathcal{X}_0$ contains an infinite subcollection of equivalent multisets, i.e., multisets containing the same elements with the same multiplicities.

We will use $\#(S, i)$ to denote the multiplicity of element $i \in [n]$ in multiset S. Choose any multiset $X \in \mathcal{X}_0$. For each $i \in [n]$, let $\mathcal{X}_1^{(i)} = \{B' \in \mathcal{X}_0 \setminus \{X\} \mid \#(B', i) < \#(X, i)\}$ be the collection of multisets of $\mathcal{X}_0$ containing fewer instances of i than X. Each $B' \in \mathcal{X}_0 \setminus \{X\}$ must exist in at least one of these collections since $X \not\subseteq B'$. But there are only n collections $\mathcal{X}_1^{(1)}, \dots, \mathcal{X}_1^{(n)}$, and $\mathcal{X}_0$ is infinite, so there must exist a collection $\mathcal{X}_1^{(i_1)}$ that is infinite. For each $0 \leq j < \#(X, i_1)$, let $\mathcal{X}_1^{(i_1,j)} = \{B' \in \mathcal{X}_1^{(i_1)} \mid \#(B', i_1) = j\}$ be the multisets in $\mathcal{X}_1^{(i_1)}$ containing exactly j instances of element i_1. Once again, there are infinitely many multisets in $\mathcal{X}_1^{(i_1)}$ but only a finite range of multiplicities j, so at least one collection $\mathcal{X}_1^{(i_1,j)}$ is infinite. Call this one $\mathcal{X}_1$.

Next define $\mathcal{X}_2$ in a similar way, but using $\mathcal{X}_1$ in place of $\mathcal{X}_0$. By our construction, every multiset in $\mathcal{X}_1$ contains the same number of instances of element $i_1 \in [n]$. When defining $\mathcal{X}_2$ then, we will have $\mathcal{X}_2^{(i_1)} = \varnothing$. Thus, there will be an element $i_2 \in [n]$ with $i_2 \neq i_1$ such that every multiset in $\mathcal{X}_2$ has the same number of instances of i_2. Since $\mathcal{X}_2 \subset \mathcal{X}_1$, every multiset in $\mathcal{X}_2$ has the same number of instances of both i_1 and i_2. If we continue to define the collections $\mathcal{X}_3, \dots, \mathcal{X}_n$ in this way, all multisets in each $\mathcal{X}_j$ will have the same numbers of instances of $i_1, \dots, i_j$. Thus, for every $X, Y \in \mathcal{X}_n$ and $i \in [n]$, $\#(X, i) = \#(Y, i)$. But then $\mathcal{X}_n$ is an infinite subcollection of equivalent multisets in $\mathcal{X}_0$, a contradiction. ◀

We now prove our superconstant memory lower bound.

▶ **Theorem 7.** *Any idle-start algorithm that solves broadcast with stabilizing termination for anonymous, synchronous, 1-interval connected dynamic networks must use $\omega(1)$ memory.*

Proof. Suppose for contradiction that $\mathcal{A}$ is an idle-start algorithm with $\Theta(1)$ space complexity that solves broadcast with stabilizing termination. Then there is a constant k such that $\mathcal{A}$ never uses more than k states. Note that throughout the rest of this proof, $f(k)$ is well-defined and has finite value by Lemma 5. We will show that the existence of this algorithm $\mathcal{A}$ contradicts Lemma 6 by constructing an infinite sequence of reachable configurations $(C_i)_{i=0}^{\infty}$ where each C_i is a disjoint union of multisets $S_i \cup T_i$ satisfying:

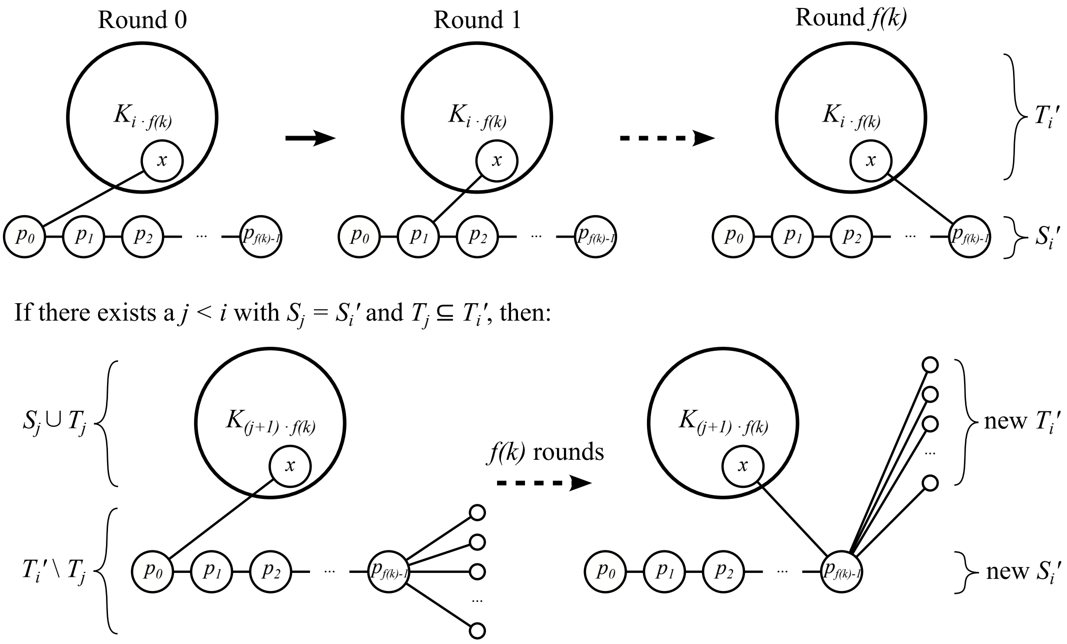


Figure 3 The time-varying graph structures used in the proof of Theorem 7.

1. $|S_i| = f(k)$,
2. $|T_i| = i \cdot f(k)$, and
3. $\neg(S_j = S_i \wedge T_j \subseteq T_i)$ for all $j \neq i$.

Initially, the broadcaster is in some state β_0 and all other nodes are in some other state $\alpha_0 \neq \beta_0$. Define the initial configuration C_0 by letting $S_0 = \{\beta_0\} \cup \{\alpha_0\}^{f(k)-1}$ and $T_0 = \varnothing$. Clearly, C_0 is reachable and satisfies the above conditions.

Now consider any $i \geq 1$ and suppose configurations $C_0, \dots, C_{i-1}$ have already been defined; we inductively define C_i as follows. Construct a time-varying graph $\mathcal{G}$ from two static components: the complete graph $K_{i \cdot f(k)}$ on $i \cdot f(k)$ nodes (including the broadcaster) and a path $p_0 p_1 \cdots p_{f(k)-1}$ (Figure 3, top). In each round $t \in \{0, \dots, f(k)-1\}$, these components are connected by a single edge $\{x, p_t\}$, where $x \in K_{i \cdot f(k)}$ is some fixed node. It can be easily shown – as we did by induction in the proof of Theorem 1 – that the nodes in $K_{i \cdot f(k)}$ send the same messages and transition to the same states in an execution of $\mathcal{A}$ on $\mathcal{G}$ as they would in an execution of $\mathcal{A}$ on $K_{i \cdot f(k)}$ by itself. By definition, these executions must stabilize on $K_{i \cdot f(k)}$ within $f(k)$ rounds. At this point, let S_i' and T_i' be the multisets of states of the nodes in the path and complete graph, respectively.

If there is no $j \in \{0, \dots, i-1\}$ such that $S_j = S_i'$ and $T_j \subseteq T_i'$, then we define configuration C_i as $S_i = S_i'$ and $T_i = T_i'$. Otherwise, if such a j does exist, we extend the execution of $\mathcal{A}$ on $\mathcal{G}$ as follows (Figure 3, bottom). Arrange the nodes of $C_j = S_j \cup T_j$ as the complete graph $K_{(j+1) \cdot f(k)}$ and any $f(k)$ nodes from $T_i' \setminus T_j$ as a path $p_0 p_1 \cdots p_{f(k)-1}$; attach the remaining nodes to the far end $p_{f(k)-1}$ of the path. Then repeat the same process as before, executing $\mathcal{A}$ for $f(k)$ rounds when these components are connected in round t by a single edge $\{x, p_t\}$, where $x \in K_{(j+1) \cdot f(k)}$ is any fixed node. Configuration C_j is reachable by induction, and again all nodes of $K_{(j+1) \cdot f(k)}$ must send the same messages and transition to the same states in the execution of $\mathcal{A}$ on $\mathcal{G}$ as they would in an execution of $\mathcal{A}$ on $K_{(j+1) \cdot f(k)}$ alone. So, by definition of $f(k)$, all nodes in $K_{(j+1) \cdot f(k)}$ must be idle after these $f(k)$ rounds. Also, $p_{f(k)-1}$

remains idle throughout this extended execution, so the initially idle non-path nodes attached to it are also idle at this time. Redefine S_i' as the $f(k)$ path nodes and T_i' as all other nodes, which as we've argued will all be idle. Again, if there is no $j \in \{0, \ldots, i-1\}$ such that $S_j = S_i'$ and $T_j \subseteq T_i'$, then define configuration C_i as $S_i = S_i'$ and $T_i = T_i'$; otherwise, extend the execution of $\mathcal{A}$ on $\mathcal{G}$ by another $f(k)$ rounds as above.

Suppose for contradiction that the condition for defining C_i is never met and the execution of $\mathcal{A}$ on $\mathcal{G}$ is extended forever. In every round of this execution, there is at least one idle node. So consider the execution of $\mathcal{A}$ on a modified $\mathcal{G}$ containing an extra node that is attached to some idle node in each round. This execution is identical to the one on $\mathcal{G}$, but the extra node would never leave its initial state α_0. Thus, there must also be non-idle nodes in every round, or else this execution stabilizes with uninformed nodes, contradicting the correctness of $\mathcal{A}$. But then this infinite execution contains non-idle nodes in every round, contradicting the supposition that $\mathcal{A}$ eventually stabilizes. Thus, multisets S_i' and T_i' with the desired condition will be found in finite time. Clearly, $C_i = S_i \cup T_i = S_i' \cup T_i'$ is reachable; we conclude by showing it satisfies the required conditions.

1. Every intermediate S_i' is defined as the states of nodes in the path components which always comprise $f(k)$ nodes. Thus, $|S_i| = f(k)$.
2. The time-varying graph $\mathcal{G}$ is defined on $|K_{i\cdot f(k)}| + |p_0 p_1 \cdots p_{f(k)-1}| = (i+1)\cdot f(k)$ nodes. Since $|S_i| = f(k)$, we have $|T_i| = i \cdot f(k)$.
3. Consider any $j \neq i$. If $j < i$ then this condition must be met since the execution defining C_i only stopped once it was true. Otherwise, if $j > i$, then by Condition 2, $|T_j| > |T_i|$ and thus $T_j \not\subseteq T_i$ is trivially true.

Thus, the infinite sequence $(C_i)_{i=0}^{\infty}$ can be defined fulfilling all of the conditions above. But this contradicts Lemma 6 stating that all such sequences are finite. ◀

5 A Logspace Algorithm for Stabilizing Termination

In this section, we shift our attention from impossibilities and lower bounds to an idle-start algorithm called Countdown that solves broadcast with stabilizing termination in our anonymous, dynamic setting. This algorithm uses $\mathcal{O}(\log n)$ memory – which is not far from the $\omega(1)$ lower bound of Theorem 7 – and stabilizes in $\mathcal{O}(n)$ rounds which is worst-case asymptotically optimal.

At a high level, the Countdown algorithm (Algorithm 1) coordinates a sequence of broadcast attempts, each lasting twice as many rounds as its predecessor until one succeeds. To facilitate these attempts, nodes store two values: `Current`, the number of rounds remaining in the current attempt; and `Maximum`, the total duration of the current attempt. In each round, non-idle nodes involved in an ongoing attempt broadcast their `Current` and `Maximum` values to their neighbors and then decrement `Current`. Idle nodes that were previously not involved in the attempt but receive these messages will join in by setting their own `Current` and `Maximum` values accordingly. This continues until messages are sent with `Current` $= 0$, indicating the end of the current attempt. If any idle node receives such a message, it detects that the broadcast should have gone on for longer. It responds by initiating a new broadcast attempt whose duration is double the previous one. These attempts continue until some attempt makes all nodes non-idle, at which point no node will initiate another attempt and the algorithm will stabilize.

Before analyzing this algorithm's correctness and complexity, we define some notation. Let $v.\texttt{var}_t$ denote the value of variable `var` in the state of node v at time t (i.e., the start of round t). In this notation, the Countdown algorithm initializes the broadcaster b with $b.\texttt{Current}_0 = 0$

Algorithm 1 Countdown for Node v.

```
    Initialization. Set Current to 0 and Maximum to 1 if $v$ is the broadcaster and both variables to $-1$ otherwise.

    Sending Messages.
 1: if Current $\neq -1$ then
 2:     Send: msg(Current, Maximum)

    State Transitions.
 3: if Current $\neq -1$ then
 4:     Current $\leftarrow$ Current $- 1$
 5: else if a message msg$(c, m)$ was received then
 6:     if $c = 0$ then                              ▷ Initiate a new attempt.
 7:         Current $\leftarrow 2m$
 8:         Maximum $\leftarrow 2m$
 9:     else if $c > 0$ then                         ▷ Join the ongoing attempt.
10:         Current $\leftarrow c - 1$
11:         Maximum $\leftarrow m$
```

and $b.\texttt{Maximum}_0 = 1$ and all other nodes $v \neq b$ with $v.\texttt{Current}_0 = v.\texttt{Maximum}_0 = -1$. Denote the set of non-idle nodes in round t as $S_t = \{v \in V : v.\texttt{Current}_t \neq -1\}$. We begin our analysis by proving that all non-idle nodes share the same `Current` and `Maximum` values.

▶ **Lemma 8.** *For all times t and any non-idle node $v \in S_t$, we have $v.\texttt{Current}_t = c_t$ and $v.\texttt{Maximum}_t = m_t$, where $c_0 = 0$, $m_0 = 1$, and*

$$(c_{t+1}, m_{t+1}) = \begin{cases} (2m_t, 2m_t) & \text{if } c_t = 0; \\ (c_t - 1, m_t) & \text{otherwise.} \end{cases}$$

Proof. Argue by induction on t. Only the broadcaster b is initially non-idle, so $c_0 = b.\texttt{Current}_0 = 0$ and $m_0 = b.\texttt{Maximum}_0 = 1$ by initialization. Now suppose the lemma holds up to and including some time $t \geq 0$ and let c_t and m_t be the unique values of $v.\texttt{Current}_t$ and $v.\texttt{Maximum}_t$ for all $v \in S_t$, respectively. Consider any node $v \in S_{t+1}$; if none exist, the lemma holds trivially. We have two cases:

1. $c_t = 0$. Suppose to the contrary that $v \in S_t$; i.e., v was also non-idle at time t. Then $v.\texttt{Current}_t = c_t = 0$. Thus, v must execute Line 4 in round t, yielding $v.\texttt{Current}_{t+1} = -1$ and becoming idle by time $t+1$, a contradiction. So v was idle at time t but became non-idle by time $t+1$, meaning it must have received one or more messages from non-idle nodes in round t. By Line 2 and the induction hypothesis, all of those messages are $\texttt{msg}(c_t = 0, m_t)$. So v must execute Lines 7–8 in round t, yielding $v.\texttt{Current}_{t+1} = v.\texttt{Maximum}_{t+1} = 2m_t$. Our choice of $v \in S_{t+1}$ was arbitrary, so $c_{t+1} = m_{t+1} = 2m_t$.
2. $c_t > 0$. First suppose $v \in S_t$. Then $v.\texttt{Current}_t = c_t > 0$, so v executes Line 4 in round t, yielding $v.\texttt{Current}_{t+1} = c_t - 1$ and $v.\texttt{Maximum}_{t+1} = m_t$ as claimed. Now suppose $v \notin S_t$. To transition from idle to non-idle in round t, v must receive one or more messages from non-idle nodes in round t. By Line 2, all of those messages are $\texttt{msg}(c_t > 0, m_t)$. So v must execute Lines 10–11, yielding $v.\texttt{Current}_{t+1} = c_t - 1$ and $v.\texttt{Maximum}_{t+1} = m_t$. Our choice of $v \in S_{t+1}$ was arbitrary, so $c_{t+1} = c_t - 1$ and $m_{t+1} = m_t$. ◀

Using the c_t and m_t values defined in Lemma 8, we next show that a broadcast attempt lasting k rounds either informs all nodes and stabilizes or involves at least $k + 1$ non-idle nodes before initiating a new attempt.

▶ **Lemma 9.** *If $k := c_t = m_t > 0$, then either (1) $S_{t+k} = V$ and $S_{t+k+1} = \varnothing$, or (2) $|S_{t+k}| \geq k+1$ and $c_{t+k+1} = m_{t+k+1} = 2k$.*

Proof. Consider any time t at which $k := c_t = m_t > 0$, the start of a new broadcast attempt by the set of nodes S_t. First suppose that for some $0 \leq i \leq k$, we have $S_{t+i} = V$; i.e., all nodes are non-idle. By Lemma 8, we have $c_{t+i} = c_t - i$. Since all nodes v are non-idle at time $t+i$ and thus have $v.\texttt{Current}_{t+i} = c_{t+i} = c_t - i$, they all execute Line 4 in round $t+i$ by decrementing $v.\texttt{Current}$. If $i < k$, then $v.\texttt{Current}_{t+i+1} = c_t - i - 1 \neq -1$, so all nodes remain non-idle and the process repeats; otherwise, if $i = k$, then $v.\texttt{Current}_{t+i+1} = c_t - k - 1 = -1$. This renders all nodes idle at time $t+k+1$, so Case 1 has occurred.

Now suppose that $S_{t+i} \neq V$ for all $0 \leq i \leq k$; i.e., there is at least one idle node throughout the attempt. We argue by induction on $0 \leq i \leq k$ that $|S_{t+i}| \geq |S_t| + i$; i.e., at least one idle node becomes non-idle in each round. The $i = 0$ case holds trivially, so suppose the claim holds up to and including some $i < k$. Since $c_{t+i} = c_t - i = k - i$ by Lemma 8, any node in S_{t+i} must remain non-idle until time $t + i + c_{t+i} = t + k$. So $|S_{t+i+1}| \geq |S_{t+i}|$. If the induction hypothesis is in fact a strict inequality, we are done:

$$|S_{t+i}| > |S_t| + i \quad \Rightarrow \quad |S_{t+i+1}| \geq |S_{t+i}| \geq |S_t| + i + 1.$$

So suppose instead that $|S_{t+i}| = |S_t| + i$. There must exist non-idle nodes at time $t+1$ since $c_{t+1} = c_t - 1 > -1$ by Lemma 8, and there must exist idle nodes at time $t+1$ by supposition. Thus, since the dynamic network is 1-interval connected, there must be some idle node $v \in V \setminus S_{t+i}$ that receives a message from a non-idle node in round $t+1$, causing v to become non-idle (Lines 7–8 or 10–11). By the induction hypothesis, $|S_{t+i+1}| \geq |S_{t+i}| + 1 = |S_t| + i + 1$.

By this induction argument, we have $|S_{t+k}| \geq |S_t| + k \geq k + 1$. By Lemma 8, we have that $c_{t+k} = c_t - k = 0$, and with another application of the same lemma, we conclude that $c_{t+k+1} = m_{t+k+1} = 2m_{t+k} = 2m_t = 2k$. So Case 2 has occurred. ◀

Our algorithm's correctness follows from the previous lemma and its time and space complexities are obtained with straightforward counting arguments.

▶ **Theorem 10.** *Countdown (Algorithm 1) correctly solves broadcast with stabilizing termination from an idle start in $\mathcal{O}(n)$ rounds and $\mathcal{O}(\log n)$ space for anonymous, synchronous, 1-interval connected dynamic networks.*

Proof. By Lemma 8, we have $c_1 = m_1 = 2$, allowing us to apply Lemma 9. But suppose to the contrary that this and all subsequent applications of the lemma result in Case 2 – where a new attempt is initiated with double the duration – and not Case 1, where all nodes are informed ($S_{t+k} = V$) and the algorithm stabilizes ($S_{t+k+1} = \varnothing$). Then there exists an attempt of duration $k \geq n$, which, by Lemma 9, ends with $|S_{t+k}| \geq k + 1 \geq n + 1$ non-idle nodes. But there are only n nodes in the network, a contradiction. So Countdown must inform all nodes and stabilize in finite time.

It remains to bound runtime and memory. Each time a new broadcast attempt is initiated, the `Maximum` variable is doubled and the algorithm runs for another `Maximum` rounds. As we already showed, once `Maximum` reaches or exceeds n, the subsequent attempt will inform all nodes and stabilize. Thus, `Maximum` doubles at most $\lceil \log_2 n \rceil$ times, meaning Countdown stabilizes in at most $\sum_{i=0}^{\lceil \log_2 n \rceil} 2^i = \mathcal{O}(2^{1+\lceil \log_2 n \rceil} - 1) = \mathcal{O}(n)$ rounds. This analysis also shows that the largest attainable `Maximum` value before its final doubling is $n-1$, so $\texttt{Maximum} \leq 2(n-1) = \mathcal{O}(n)$. Since $-1 \leq \texttt{Current} \leq \texttt{Maximum}$, we also have $\texttt{Current} = \mathcal{O}(n)$, implying that Countdown has $\mathcal{O}(\log n)$ space complexity. ◀

6 Conclusion

This paper investigated what memory is necessary for anonymous, synchronous, 1-interval connected dynamic networks to deterministically solve broadcast with some termination conditions. We considered both *termination detection* where the broadcaster must eventually declare that every node has been informed and *stabilizing termination* where nodes must eventually stop sending messages. Combining our results with the established literature, we now know the following about this problem:

- *Termination Detection.* Regardless of memory, broadcast with termination detection is impossible for idle-start algorithms (Theorem 1) and for non-idle-start algorithms when the number of broadcasters is unknown (Theorem 2). Any (non-idle-start) algorithm solving broadcast with termination detection must use $\Omega(\log n)$ memory per node (Theorem 3). The best known space complexity for this problem follows from Di Luna and Viglietta's history trees algorithm which uses $\mathcal{O}(n^3 \log n)$ memory in the worst case [16].
- *Stabilizing Termination.* Any idle-start algorithm solving broadcast with stabilizing termination must use $\omega(1)$ memory per node (Theorem 7). As a positive result, this problem is solvable with logarithmic memory under standard synchrony: Countdown is a $\mathcal{O}(\log n)$ memory, linear time algorithm achieving stabilizing termination without identifiers or knowledge of n (Theorem 10).

For stabilizing termination, our $\omega(1)$ memory bound holds only for idle-start algorithms and our $\mathcal{O}(\log n)$ memory Countdown algorithm happens to be idle-start. In the non-idle-start regime where non-broadcaster nodes can send messages from their initial states, can we obtain a sublogarithmic space algorithm? Can any lower bound be shown? The contradiction at the heart of our lower bound technique for idle-start algorithms identified configurations C_i that are reachable with or without an extra uninformed node. In the non-idle-start case, however, this extra uninformed node may send new and unaccounted for messages and we can no longer guarantee C_i will still be reached, requiring a different approach.

The $\Omega(\log n)$ and $\mathcal{O}(n^3 \log n)$ memory bounds for termination detection leave open a significant gap for further improvement. Our logarithmic lower bound shows that termination detection requires enough memory to count to n, and indeed there is a straightforward solution for termination detection if (an upper bound on) n can be obtained: simply wait for n rounds after broadcasting information for the first time and then declare all nodes have been informed. Can broadcast with termination detection be achieved without solving exact counting? If not, what approaches could yield algorithms for exact counting that are more space-efficient than history trees? Viglietta recently proposed the existence of logspace counting algorithms as an open problem unlikely to be solved by history trees [37]; we are unsure such algorithms exist at all, as we suspect our $\Omega(\log n)$ bound can be improved.

Finally, we note that all impossibility results and lower bounds in this paper apply also to any problem broadcast reduces to, such as exact counting. Thus, these results and any future improvements shed important light on the requirements of nontrivial terminating computation in anonymous dynamic networks.

References

1 Karine Altisen, Stéphane Devismes, Anaïs Durand, Colette Johnen, and Franck Petit. Self-Stabilizing Systems in Spite of High Dynamics. *Theoretical Computer Science*, 964:113966, 2023. doi:10.1016/j.tcs.2023.113966.

2 David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 131–145, Banff, AB, Canada, 2001. ACM. doi:10.1145/502034.502048.

3 John Augustine, Gopal Pandurangan, and Peter Robinson. Distributed Algorithmic Foundations of Dynamic Networks. *ACM SIGACT News*, 47(1):30, 2016.

4 Alysson Bessani, Eduardo Alchieri, João Sousa, André Oliveira, and Fernando Pedone. From Byzantine Replication to Blockchain: Consensus is Only the Beginning. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 424–436, Valencia, Spain, 2020. IEEE. doi:10.1109/DSN48063.2020.00057.

5 Arnaud Casteigts. *A Journey through Dynamic Networks (with Excursions)*. Habilitation à diriger des recherches, University of Bordeaux, Bordeaux, France, 2018.

6 Arnaud Casteigts, Paola Flocchini, Bernard Mans, and Nicola Santoro. Deterministic Computations in Time-Varying Graphs: Broadcasting under Unstructured Mobility. In *Theoretical Computer Science*, volume 323 of *IFIP Advances in Information and Communication Technology*, pages 111–124, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-15240-5_9.

7 Arnaud Casteigts, Paola Flocchini, Bernard Mans, and Nicola Santoro. Shortest, Fastest, and Foremost Broadcast in Dynamic Networks. *International Journal of Foundations of Computer Science*, 26(04):499–522, 2015. doi:10.1142/S0129054115500288.

8 Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-Varying Graphs and Dynamic Networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012. doi:10.1080/17445760.2012.668546.

9 Maitri Chakraborty, Alessia Milani, and Miguel A. Mosteiro. A Faster Exact-Counting Protocol for Anonymous Dynamic Networks. *Algorithmica*, 80(11):3023–3049, 2018. doi:10.1007/s00453-017-0367-4.

10 Arjun Chandrasekhar, Deborah M. Gordon, and Saket Navlakha. A Distributed Algorithm to Maintain and Repair the Trail Networks of Arboreal Ants. *Scientific Reports*, 8(1):9297, 2018. doi:10.1038/s41598-018-27160-3.

11 Giuseppe Di Luna and Roberto Baldoni. Non Trivial Computations in Anonymous Dynamic Networks. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, volume 46 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33:1–33:16, Rennes, France, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.OPODIS.2015.33.

12 Giuseppe A. Di Luna and Giovanni Viglietta. Computing in Anonymous Dynamic Networks Is Linear. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1122–1133, Denver, CO, USA, 2022. IEEE. doi:10.1109/FOCS54457.2022.00108.

13 Giuseppe A. Di Luna and Giovanni Viglietta. Optimal Computation in Leaderless and Multi-Leader Disconnected Anonymous Dynamic Networks. In *37th International Symposium on Distributed Computing (DISC 2023)*, volume 281 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:20, L'Aquila, Italy, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2023.18.

14 Giuseppe Antonio Di Luna, Roberto Baldoni, Silvia Bonomi, and Ioannis Chatzigiannakis. Conscious and Unconscious Counting on Anonymous Dynamic Networks. In *Distributed Computing and Networking*, volume 8314 of *Lecture Notes in Computer Science*, pages 257–271, Berlin, Heidelberg, 2014. Springer. doi:10.1007/978-3-642-45249-9_17.

15 Giuseppe Antonio Di Luna, Roberto Baldoni, Silvia Bonomi, and Ioannis Chatzigiannakis. Counting in Anonymous Dynamic Networks under Worst-Case Adversary. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 338–347, Madrid, Spain, 2014. IEEE. doi:10.1109/ICDCS.2014.42.

16 Giuseppe Antonio Di Luna and Giovanni Viglietta. Brief Announcement: Efficient Computation in Congested Anonymous Dynamic Networks. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, pages 176–179, Orlando, FL, USA, 2023. ACM. doi:10.1145/3583668.3594590.

17 Michael Dinitz, Jeremy Fineman, Seth Gilbert, and Calvin Newport. Smoothed Analysis of Information Spreading in Dynamic Networks. In *36th International Symposium on Distributed Computing (DISC 2022)*, volume 246 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:22, Augusta, GA, USA, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2022.18.

18 Michael Dinitz, Jeremy T. Fineman, Seth Gilbert, and Calvin Newport. Smoothed Analysis of Dynamic Networks. *Distributed Computing*, 31(4):273–287, 2018. doi:10.1007/s00446-017-0300-8.

19 Michael Feldmann, Christian Scheideler, and Stefan Schmid. Survey on Algorithms for Self-Stabilizing Overlay Networks. *ACM Computing Surveys*, 53(4):74:1–74:24, 2020. doi:10.1145/3397190.

20 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors. *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*. Springer, Cham, 2019. doi:10.1007/978-3-030-11072-7.

21 Vincent Gramoli. From Blockchain Consensus Back to Byzantine Consensus. *Future Generation Computer Systems*, 107:760–769, 2020. doi:10.1016/j.future.2017.09.023.

22 Heiko Hamann. *Swarm Robotics: A Formal Approach*. Springer, Cham, 1 edition, 2018. doi:10.1007/978-3-319-74528-2.

23 Walter Hussak and Amitabh Trehan. On Termination of a Flooding Process. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 153–155, Toronto, ON, Canada, 2019. ACM. doi:10.1145/3293611.3331586.

24 Walter Hussak and Amitabh Trehan. On the Termination of Flooding. In *37th International Symposium on Theoretical Aspects of Computer Science (STACS 2020)*, volume 154 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:3, Montpellier, France, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.STACS.2020.17.

25 Walter Hussak and Amitabh Trehan. Termination of Amnesiac Flooding. *Distributed Computing*, 36(2):193–207, 2023. doi:10.1007/s00446-023-00448-y.

26 Dariusz R. Kowalski and Miguel A. Mosteiro. Polynomial Counting in Anonymous Dynamic Networks with Applications to Anonymous Dynamic Algebraic Computations. *Journal of the ACM*, 67(2):1–17, 2020. doi:10.1145/3385075.

27 Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed Computation in Dynamic Networks. In *Proceedings of the 42nd ACM Symposium on Theory of Computing*, pages 513–522, Cambridge, MA, 2010. ACM. doi:10.1145/1806689.1806760.

28 Jintao Liu, Arthur Prindle, Jacqueline Humphries, Marçal Gabalda-Sagarra, Munehiro Asally, Dong-yeon D. Lee, San Ly, Jordi Garcia-Ojalvo, and Gürol M. Süel. Metabolic Co-Dependence Gives Rise to Collective Oscillations within Biofilms. *Nature*, 523(7562):550–554, 2015. doi:10.1038/nature14660.

29 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

30 Othon Michail, Ioannis Chatzigiannakis, and Paul G. Spirakis. Naming and Counting in Anonymous Unknown Dynamic Networks. In *Stabilization, Safety, and Security of Distributed Systems*, volume 8255 of *Lecture Notes in Computer Science*, pages 281–295, Cham, 2013. Springer. doi:10.1007/978-3-319-03089-0_20.

31 Regina O'Dell and Roger Wattenhofer. Information Dissemination in Highly Dynamic Graphs. In *Proceedings of the 2005 Joint Workshop on Foundations of Mobile Computing*, pages 104–110, Cologne, Germany, 2005. ACM. doi:10.1145/1080810.1080828.

32 Shunhao Oh, Dana Randall, and Andréa W. Richa. Adaptive Collective Responses to Local Stimuli in Anonymous Dynamic Networks. In *2nd Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2023)*, volume 257 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:23, Pisa, Italy, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SAND.2023.6.

33 Arthur Prindle, Jintao Liu, Munehiro Asally, San Ly, Jordi Garcia-Ojalvo, and Gürol M. Süel. Ion Channels Enable Electrical Communication in Bacterial Communities. *Nature*, 527(7576):59–63, 2015. doi:10.1038/nature15709.

34 Michel Raynal, Julien Stainer, Jiannong Cao, and Weigang Wu. A Simple Broadcast Algorithm for Recurrent Dynamic Systems. In *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, pages 933–939, Victoria, BC, Canada, 2014. IEEE. doi:10.1109/AINA.2014.115.

35 Leonie Reichert, Samuel Brack, and Björn Scheuermann. A Survey of Automatic Contact Tracing Approaches Using Bluetooth Low Energy. *ACM Transactions on Computing for Healthcare*, 2(2):1–33, 2021. doi:10.1145/3444847.

36 Corina E. Tarnita, Alex Washburne, Ricardo Martinez-Garcia, Allyson E. Sgro, and Simon A. Levin. Fitness Tradeoffs between Spores and Nonaggregating Cells Can Explain the Coexistence of Diverse Genotypes in Cellular Slime Molds. *Proceedings of the National Academy of Sciences*, 112(9):2776–2781, 2015. doi:10.1073/pnas.1424242112.

37 Giovanni Viglietta. History Trees and Their Applications. In *Structural Information and Communication Complexity*, volume 14662 of *Lecture Notes in Computer Science*, pages 3–23, Cham, 2024. Springer. doi:10.1007/978-3-031-60603-8_1.

Connectivity Labeling in Faulty Colored Graphs

Asaf Petruschka
Weizmann Institute of Science, Rehovot, Israel

Shay Spair
Weizmann Institute of Science, Rehovot, Israel

Elad Tzalik
Weizmann Institute of Science, Rehovot, Israel

Abstract

Fault-tolerant connectivity labelings are schemes that, given an n-vertex graph $G = (V, E)$ and a parameter f, produce succinct yet informative labels for the elements of the graph. Given *only the labels* of two vertices u, v and of the elements in a faulty-set F with $|F| \leq f$, one can determine if u, v are connected in $G - F$, the surviving graph after removing F. For the edge or vertex faults models, i.e., $F \subseteq E$ or $F \subseteq V$, a sequence of recent work established schemes with $\text{poly}(f, \log n)$-bit labels for general graphs. This paper considers the *color faults* model, recently introduced in the context of spanners [Petruschka, Sapir and Tzalik, ITCS '24], which accounts for known correlations between failures. Here, the edges (or vertices) of the input G are arbitrarily colored, and the faulty elements in F are colors; a failing color causes all edges (vertices) of that color to crash. While treating color faults by naïvly applying solutions for many failing edges or vertices is inefficient, the known correlations could potentially be exploited to provide better solutions.

Our main contribution is settling the label length complexity for connectivity under one color fault ($f = 1$). The existing implicit solution, by black-box application of the state-of-the-art scheme for edge faults of [Dory and Parter, PODC '21], might yield labels of $\Omega(n)$ bits. We provide a deterministic scheme with labels of $\tilde{O}(\sqrt{n})$ bits in the worst case, and a matching lower bound. Moreover, our scheme is *universally optimal*: even schemes tailored to handle only colorings of one specific graph topology (i.e., may store the topology "for free") cannot produce asymptotically smaller labels. We characterize the optimal length by a new graph parameter $\texttt{bp}(G)$ called the *ball packing number*. We further extend our labeling approach to yield a routing scheme avoiding a single forbidden color, with routing tables of size $\tilde{O}(\texttt{bp}(G))$ bits. We also consider the *centralized* setting, and show an $\tilde{O}(n)$-space oracle, answering connectivity queries under one color fault in $\tilde{O}(1)$ time. Curiously, by our results, no oracle with such space can be *evenly* distributed as labels.

Turning to $f \geq 2$ color faults, we give a randomized labeling scheme with $\tilde{O}(n^{1-1/2^f})$-bit labels, along with a lower bound of $\Omega(n^{1-1/(f+1)})$ bits. For $f = 2$, we make partial improvement by providing labels of $\tilde{O}(\text{diam}(G)\sqrt{n})$ bits, and show that this scheme is (nearly) optimal when $\text{diam}(G) = \tilde{O}(1)$.

Additionally, we present a general reduction from the above *all-pairs* formulation of fault-tolerant connectivity labeling (in any fault model) to the *single-source* variant, which could also be applicable for centralized oracles, streaming, or dynamic algorithms.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Data structures design and analysis

Keywords and phrases Labeling schemes, Fault-tolerance

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.36

Related Version *Full Version*: `https://arxiv.org/abs/2402.12144`

Funding *Asaf Petruschka*: This work is supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, grant agreement No. 949083, and by the Israeli Science Foundation (ISF), grant 2084/18.
Shay Spair: Partially supported by the Israeli Council for Higher Education (CHE) via the Weizmann Data Science Research Center.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 36; pp. 36:1–36:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Acknowledgements We are grateful to Merav Parter for encouraging this collaboration, and for helpful guidance and discussions.

1 Introduction

Labeling schemes are important distributed graph data structures with diverse applications in graph algorithms and distributed computing, concerned with assigning the vertices of a graph (and possibly also other elements, such as edges) with succinct yet informative *labels*. Many real-life networks are often error-prone by nature, which motivates the study of fault-tolerant graph structures and services. In a fault-tolerant connectivity labeling scheme, we are given an n-vertex graph $G = (V, E)$ and an integer f, and should assign short labels to the elements of G, such that the following holds: For every pair of vertices $u, v \in V$ and faulty-set F with $|F| \leq f$, one can determine if u and v are connected in $G - F$ by merely inspecting the labels of the elements in $\{u, v\} \cup F$. The main complexity measure is the maximal *label length* (in bits), while construction and query time are secondary measures.

The concept of edge/vertex-fault-tolerant labeling, aka *forbidden set* labeling, was explicitly introduced by Courcelle and Twigg [9]. Earlier work on fault-tolerant connectivity and distance labeling focused on graph families such as planar graphs and graphs with bounded treewidth or doubling dimension [9, 8, 1, 2]. Up until recently, designing edge- or vertex-fault-tolerant connectivity labels for general graphs remained fairly open. Dory and Parter [10] were the first to construct *randomized* labeling schemes for connectivity under f *edge* faults, where a query is answered correctly with high probability,[1] with length of $O(\min\{f + \log n, \log^3 n\})$ bits. Izumi, Emek, Wadayama and Masuzawa [21] derandomized this construction, showing *deterministic* labels of $\tilde{O}(f^2)$ bits.[2] Turning to f *vertex* faults, Parter and Petruschka [32] designed connectivity labels for $f \leq 2$ with $\tilde{O}(1)$ bits. Very recently, Parter, Petruschka and Pettie [33] provided a randomized scheme for f vertex faults with $\tilde{O}(f^3)$ bits and a derandomized version with $\tilde{O}(f^7)$ bits, along with a lower bound of $\Omega(f)$ bits. Another important research area which is closely related to fault-tolerant labeling concerns the design of *forbidden-set routing schemes*, see e.g. [9, 1, 2, 10, 33]. (Further background on such routing schemes is given in Section 1.1.1.)

In this work, we consider labeling schemes for connectivity under *color faults*, a model that was very recently introduced in the context of graph spanners [34], which intuitively accounts for known correlations between failures. In this model, the edges or vertices of the input graph G are arbitrarily partitioned into classes, or equivalently, associated with *colors*, and a set F of f such color classes might fail. A failing color causes all edges (vertices) of that color to crash.The surviving subgraph $G - F$ is formed by deleting every edge[3] or vertex with color from F. The scheme must assign labels to the vertices *and to the colors* of G, so that a connectivity query $\langle u, v, F \rangle$ can be answered by inspecting only the labels of the vertices u, v and of the colors in F.

This new notion generalizes edge/vertex fault-tolerant schemes, that are obtained in the special case when each edge or vertex has a unique color. However, in the general case, even a single color fault may correspond to many and arbitrarily spread edge/vertex faults, which poses a major challenge. Tackling this issue by naively applying the existing solutions for

[1] Throughout, the term with high probability (w.h.p.) stands for probability at least $1 - 1/n^{\alpha}$, where $\alpha > 0$ is a constant that can be made arbitrarily large through increasing the relevant complexity measure by a constant factor.

[2] Throughout, the $\tilde{O}(\cdot)$ notation hides polylog(fn) factors.

[3] In the edge-colored case we naturally allow multi-graphs where parallel edges may have different colors.

many individual edge/vertex faults (i.e., by letting the label of a color store all labels given to elements in its class) may result in very large labels of $\Omega(n)$ bits or more, even when $f = 1$. On a high level, this work shows that the correlation between the faulty edges/vertices, predetermined by the colors, can be used to construct much better solutions.

Related Work on Colored Graphs. Faulty colored classes have been used to model Shared Risk Resource Groups (SRRG) in optical telecommunication networks, multi-layered networks, and various other practical contexts; see [7, 24, 40] and the references therein. Previous work mainly focused on centralized algorithms for colored variants of classical graph problems (and their hardness). A notable such problem is diverse routing, where the goal is to find two (or more) color disjoint paths between two vertices [20, 12, 27]. Another is the colored variant of minimum cut, known also as the *hedge connectivity*, where the objective is to determine the minimum number of colors (aka hedges) whose removal disconnects the graph; see e.g. [16, 39, 14].

A different line of work focuses on distances to or between color classes, and specifically on (centralized) data structures that, given a query $\langle v, c\rangle$, report the closest c-colored vertex to v in the graph, or the (approximate) distance from it [19, 5, 26, 15, 36, 13].

Remark on Color Lists. One might ask what happens if the correlated sets of failures are allowed to have some “bounded” overlap. This can be modeled by *color lists*: every vertex/edge has a small list of associated colors, and the failure of any one of them will cause its crash. In some problems, the relevant complexity measures might be affected even when lists only have some constant size, see e.g. [34]. However, all the results of the current paper can be easily shown to hold with color lists of constant size, and we therefore focus only on the disjoint color classes model.

1.1 Our Results

We initiate the study of fault-tolerant labeling schemes in colored graphs. All of our results apply both to edge-colored and to vertex-colored (multi)-graphs.

1.1.1 Single Color Fault ($f = 1$)

For $f = 1$, i.e., a single faulty color, we (nearly) settle the complexity of the problem, by showing a simple construction of labels with length $O(\sqrt{n}\log n)$ bits, along with a matching lower bound of $\Omega(\sqrt{n})$ bits. In fact, our scheme provides a strong beyond worst-case guarantee: for every given graph G, the length of the assigned labels is (nearly) the best possible, even compared to schemes that are tailor-made to only handle colorings of the topology in G,or equivalently, are allowed to store the uncolored topology “for free” in all the labels. (By the *topology* of G, we mean the uncolored graph obtained from G by ignoring the colors. Slightly abusing notation, we refer to this object as *the graph topology* G, rather than the colored graph G.) Guarantees of this form, known as *universal optimality*, have sparked major interest in the graph algorithms community, and particularly in recent years, following the influential work of Haeupler, Wajc and Zuzic [18] in the distributed setting. On an intuitive level, the universal optimality implies that even when restricting attention to any class of graphs, e.g. planar graphs, our scheme performs asymptotically as well as the optimal scheme for this specific class. We note that one cannot compete with a scheme that is optimal for the given graph *and* its coloring (aka “instance optimal”), as such a tailor-made scheme may store the entire colored graph “for free”, and the labels merely need to specify the query.

Our universally optimal labels are based on a new graph parameter called the *ball packing number*, denoted by $\mathsf{bp}(G)$. When disregarding minor nuances by assuming G is connected, $\mathsf{bp}(G)$ is the maximum integer r such that one can fit r disjoint balls of radius r in the topology of G (see formal definition in Section 3.1). The ball packing number of an n-vertex graph is always at most $\sqrt{n}$, but often much smaller. For example, $\mathsf{bp}(G)$ is smaller than the *diameter* of G. In Section 3, we show the following:

▶ **Theorem 1** ($f = 1$, informal)**.** *There is a connectivity labeling scheme for one color fault, that for every n-vertex graph G, assigns $O(\mathsf{bp}(G) \log n)$-bit labels. Moreover, $\Omega(\mathsf{bp}(G))$-bit labels are necessary, even for labeling schemes tailor-made for the topology of G, i.e., where the uncolored topology is given in addition to the query labels.*

The lower bound in Theorem 1 is information-theoretic, obtained via communication complexity. The upper bound is based on observing that (when G is connected) there is a subset A of $O(\mathsf{bp}(G))$ vertices which is $O(\mathsf{bp}(G))$-*ruling*: every vertex in G has a path to A of length $O(\mathsf{bp}(G))$.

Routing Schemes. Building upon our labeling scheme, we additionally provide a *routing scheme* for avoiding any single forbidden color. This is a natural extension of the forbidden-set routing framework, initially introduced by [9] (see also [1, 2, 10, 33]), to the setting of colored graphs. We refer the reader to [10] for an overview of forbidden-set routing, and related settings. Such a routing scheme consists of two algorithms. The first is a preprocessing (centralized) algorithm that computes *routing tables* to be stored at each vertex of G, and *labels* for the vertices and the colors. The second is a distributed routing algorithm that enables routing a message M from a source vertex s to a target vertex t avoiding edges of color c. Initially, the labels of s, t, c are found in the source s. Then, at each intermediate node v in the route, v should use the information in its table, and in the (short) header of the message, to determine where the message should be sent; formally, v should compute the *port number* of the next edge to be taken from v (which must not be of color c). It may also edit the header for future purposes. The main concern is minimizing the size of the tables and labels, and even more so of the header (as it is communicated through the route). We show:

▶ **Theorem 2.** *There is a deterministic routing scheme for avoiding one forbidden color such that, for a given colored n-vertex graph G, the following hold:*

- *The routing tables stored at the vertices are all of size $O(\mathsf{bp}(G) \log n)$ bits.*
- *The labels assigned to the vertices and the colors are of size $O(\mathsf{bp}(G) \log n)$ bits.*
- *The header size required for routing a message is of $O(\log n)$ bits.*

Another important concern is optimizing the *stretch* of the routing scheme, which is the ratio between the length of the routing path and the length of the shortest s, t path in $G - c$. Unfortunately, our routing scheme does not provide good stretch guarantees, and optimizing it is an interesting direction for future work. We note, however, that the need to avoid edges of color c by itself poses a nontrivial challenge, and black-box application of the state-of-the-art routings schemes of Dory and Parter [10] for avoiding $f = \Omega(n)$ individual edges would yield large labels, tables and headers, *and* large stretch (all become $\Omega(n)$).

Centralized Oracles. We end our discussion for $f = 1$ by considering *centralized oracles* (data structures) for connectivity under a single color fault. In this setting, one can utilize centralization to improve on the naive approach of storing all labels. We note that this problem can be solved using existing $O(n)$-space and $O(\log\log n)$-query time oracles for *nearest colored ancestor* on trees [28, 15], yielding the same bounds for single color fault connectivity oracles. Interestingly, our lower bound shows that oracles with such space cannot be evenly distributed into labels.

1.1.2 f Color Faults

It has been widely noted that in fault-tolerant settings, handling even two faults may be significantly more challenging than handling a single fault. Such phenomena appeared, e.g., in distance oracles [11], min-cut oracles [3], reachability oracles [6] and distance preservers [29, 17, 30]. In our case, this is manifested in generalized upper and lower bounds on the label length required to support f color faults, exhibiting a gap when $f \geq 2$; our upper bound is roughly $\tilde{O}(n^{1-1/2^f})$ bits, while the lower bound is $\Omega(n^{1-1/(f+1)})$ bits (both equal $\tilde{\Theta}(\sqrt{n})$ when $f = 1$).

▶ **Theorem 3** ($f \geq 2$ upper bound, informal)**.** *There is a randomized labeling scheme for connectivity under f color faults with label length of $\min\{fn^{1-1/2^f}, n\} \cdot \mathrm{polylog}(fn)$ bits.*

▶ **Theorem 4** ($f \geq 2$ lower bound, informal)**.** *A labeling scheme for connectivity under f color faults must have label length of $\Omega(n^{1-1/(f+1)})$ bits for constant f, hence $\Omega(n^{1-o(1)})$ bits for $f = \omega(1)$.*

Curiously, in the seemingly unrelated problem of small-size fault-tolerant distance preservers (FT-BFS) introduced by Parter and Peleg [31], there is a similar gap in the known bounds for $f \geq 3$, of $O(n^{2-1/2^f})$ and $\Omega(n^{2-1/(f+1)})$ edges [29, 4]. Notably, for the case of $f = 2$, Parter [29] provided a tight upper bound of $O(n^{5/3})$, later simplified by Gupta and Khan [17]. Revealing connections between FT-BFS structures and the labels problem of this paper is an intriguing direction for future work.

Apart from the gap between the bounds, there are a few more noteworthy differences from the case of a single color fault. First, the scheme of Theorem 3 is *randomized*, as opposed to the deterministic scheme for $f = 1$ (Theorem 1). Moreover, the construction is based on different techniques, combining three main ingredients: (1) sparsification tools for colored graphs [34], (2) the (randomized) edge fault-tolerant labeling scheme of [10], and (3) a recursive approach of [32]. Second, the lower bound of Theorem 4 is *existential* (but still information-theoretic): it relies on choosing a fixed "worst-case" graph topology, and encoding information by coloring it and storing some of the resulting labels. We further argue that this technique cannot yield a lower bound stronger than $\tilde{\Omega}(n^{1-1/(f+1)})$ bits. This is due to the observation that a color whose label is not stored can be considered *never faulty*, combined with the existence of efficient labeling schemes when the number of colors is small.

For the special case of two color faults, we provide another scheme, with label length of $\tilde{O}(D\sqrt{n})$ bits for graphs of diameter at most D.

▶ **Theorem 5** ($f = 2$ upper bound, informal)**.** *There is a labeling scheme for connectivity under two color faults with label length of $\tilde{O}(D\sqrt{n})$ bits.*

This beats the general scheme when $D = O(n^{1/4-\epsilon})$, and demonstrates that the existential $\Omega(n^{2/3})$ lower bound does not apply to graphs with diameter $D = O(n^{1/6-\epsilon})$. Further, this scheme is existentially optimal (up to logarithmic factors) for graphs with $D = \tilde{O}(1)$. We hope this construction could serve as a stepping stone towards closing the current gap between our bounds, and towards generalizing $\mathsf{bp}(G)$ for the case of $f = 2$. Table 1 summarizes our main results on connectivity labeling under color faults.

Table 1 A summary of our results on f color fault-tolerant connectivity labeling schemes. The table shows the provided length bounds (in bits) for such schemes.

No. faults	Upper bound		Lower bound	
$f = 1$	$\tilde{O}(\mathsf{bp}(G)) = \tilde{O}(\sqrt{n})$	Thm 1	$\Omega(\mathsf{bp}(G))$ ($\Omega(\sqrt{n})$ in worst-case)	Thm 1
$f = 2$	$\tilde{O}(\mathrm{diam}(G)\sqrt{n})$	Thm 5	$\Omega(n^{2/3})$	Thm 4
	$\tilde{O}(n^{3/4})$	Thm 3		
$f = O(1)$	$\tilde{O}(n^{1-1/2^f})$		$\Omega(n^{1-1/(f+1)})$	
$f = \omega(1)$	$\tilde{O}(n)$		$\Omega(n^{1-o(1)})$	

Due to space limitations, the full discussion and formal proofs for $f \geq 2$ color faults are deferred to the full version of the paper.

1.1.3 Equivalence Between All-Pairs and Single-Source Connectivity

In the *single-source* variant of fault-tolerant connectivity, given are an n-vertex graph G with a designated *source vertex* s, and an integer f. It is then required to support queries of the form $\langle u, F\rangle$, where $u \in V$ and F is a faulty-set of size at most f, by reporting whether u is connected to s in $G - F$. Here, and throughout this discussion, we do not care about the type of faulty elements; these could be edges, vertices or colors. For concreteness, we focus our discussion on labeling schemes, although it applies more generally to other models, e.g., centralized oracles, streaming, and dynamic algorithms. Clearly, every labeling scheme for *all-pairs* fault-tolerant connectivity can be transformed into a single-source variant by including s's label in all other labels, which at most doubles the label length. We consider the converse direction, and show that a single-source scheme can be used as a black-box to obtain an all-pairs scheme with only a small overhead in length.

▶ **Theorem 6** (Single-source reduction, informal). *Suppose there is a single-source f fault-tolerant connectivity labeling scheme using labels of at most $b(n, f)$ bits. Then, there is an all-pairs f fault-tolerant connectivity labeling scheme with $\tilde{O}(b(n+1, f))$-bit labels.*

The reduction is based on the following idea. Suppose we add a new source vertex s to G, and include each edge from s to the other vertices independently with probability p. Given a query $\langle u, v, F\rangle$, if u, v are originally connected in $G - F$, they must agree on connectivity to the new source s, regardless of p. However, if u, v are disconnected in $G - F$, and p is such that $1/p$ is roughly the size of u's connected component in $G - F$, then with constant probability, u and v will disagree on connectivity to s. The full proof appears in Section 4.

1.2 Discussion and Future Directions

While our work provides an essentially complete picture for the case of a single color fault, our results for $f \geq 2$ color faults still leave open many interesting directions for future research:

- Can we close the gap between the $\tilde{O}(n^{1-1/2^f})$ and $\Omega(n^{1-1/(f+1)})$ bounds? Concretely, is there a labeling scheme for connectivity under $f = 2$ color faults with labels of $\tilde{O}(n^{2/3})$ bits? Can our solution for low-diameter graphs be utilized to obtain such a scheme?
- Is there a graph parameter that generalizes $\mathsf{bp}(G)$ and characterizes the length of a universally optimal labeling scheme for $f \geq 2$? Notably, even very simple graphs with small diameter and ball packing number admit a lower bound of $\Omega(\sqrt{n})$ bits for $f = 2$ (as shown in the full version).

- Can we provide non-trivial *centralized oracles* for connectivity under $f \geq 2$ color faults?
- Are there routing schemes for avoiding $f \geq 2$ forbidden colors with small header size? Our labeling scheme for $f \geq 2$ could be extended to such a routing scheme, but with a large header size of $\tilde{O}(n^{1-1/2^f})$ bits.

Another intriguing direction is going beyond connectivity queries; a natural goal is to additionally obtain approximate distances, which is open even for $f = 1$. This problem is closely related to providing forbidden color routing schemes with good stretch guarantees.

2 Preliminaries

Colored Graphs. Throughout, we denote the given input graph by G, which is an undirected graph with n vertices $V = V(G)$, and m edges $E = E(G)$. The graph G may be a multi-graph, i.e., there may be several different edges with the same endpoints (parallel edges). The edges or the vertices of G are each given a color from a set of C possible colors. The coloring is *arbitrary*; there are no "legality" restrictions (e.g., edges sharing an endpoint may have the same color). Without loss of generality, we sometimes assume that $C \leq \max\{m, n\}$, and that the set of colors is $[C]$. For a (faulty) subset of colors F, we denote by $G - F$ the subgraph of G where all edges (or vertices) with color from F are deleted. When F is a singleton $F = \{c\}$, we use the shorthand $G - c$.

In some cases, we refer only to the *topology* of the graph, and ignore the coloring. Put differently, we sometimes consider the family of inputs given by all different colorings of a fixed graph. This object is referred to as the graph *topology* G, rather than the graph G. We denote by $\mathrm{dist}_G(u, v)$ the number of edges in a u-v shortest path (and ∞ if no such path exist). For a non-empty $A \subseteq V$, the distance from $u \in V$ to A is defined as $\mathrm{dist}_G(u, A) = \min\{\mathrm{dist}_G(u, a) \mid a \in A\}$.

Our presentation focuses, somewhat arbitrarily, on the *edge-colored* case; throughout, this case is assumed to hold unless we explicitly state otherwise. This is justified by the following discussion.

Vertex vs. Edge Colorings. An edge-colored instance can be reduced to a vertex-colored one, and vice versa, by subdividing each edge[4] $e = \{u, v\}$ into two edges $\{u, x_e\}$ and $\{x_e, v\}$, where x_e is a new vertex. If the original instance has edge colors, we give the new instance vertex colors, by coloring each new vertex x_e with the original color of the edge e. (The original vertices get a new "never-failing" color.) For the other direction, we color each of $\{u, x_e\}$ and $\{x_e, v\}$ by the color of the original vertex incident to it, i.e., $\{u, x_e\}$ gets u's color, and $\{x_e, v\}$ gets v's color.

These easy reductions increase the number of vertices to $n + m$, which a prioi might seem problematic. However, as shown by [34], given any fixed (constant) bound f on the number of faulty colors, one can replace a given input instance (either vertex- or edge-colored) by an equivalent sparse subgraph with only $\tilde{O}(n)$ edges, that has the same connectivity as the original graph under any set of at most f color faults. So, by sparsifying before applying the reduction, the number of vertices increases only to $\tilde{O}(n)$. Moreover, all our results translate rather seamlessly between the edge-colored and the vertex-colored cases, even without the general reductions presented above (we explain this separately for each result).

[4] Throughout, we slightly abuse notation and write $e = \{u, v\}$ to say that e has endpoints u, v, even though there might be several different edges with these endpoints.

Vertex and Component IDs. We assume w.l.o.g. that the vertices have unique $O(\log n)$-bit identifiers from $[n]$, where $\mathsf{id}(v)$ denotes the identifier of $v \in V$. Using these, we define identifiers for connected components in subgraphs of G, as follows. When G' is a subgraph of G and $v \in V(G')$, we define $\mathsf{cid}(v, G') = \min\{\mathsf{id}(u) \mid u, v \text{ connected in } G'\}$. This ensures $\mathsf{cid}(u, G') = \mathsf{cid}(v, G')$ iff u, v are in the same connected component in G'. Therefore, if one can compute $\mathsf{cid}(v, G - F)$ from the labels of v, F, then, using the same labels, one can answer connectivity queries subject to faults.

Indexing Lower Bound. Our lower bounds rely on the classic *indexing* lower bound from communication complexity. In the one-way communication problem $\textsc{Index}(N)$, Alice holds a string $x \in \{0,1\}^N$, and Bob holds an index $i \in [0, N-1]$. The goal is for Alice to send a message to Bob, such that Bob can recover x_i, the i-th bit of x. Crucially, the communication is one-way; Bob cannot send any message to Alice. The protocols are allowed to be randomized, in which case both Alice and Bob have access to a public random string. The following lower bound on the number of bits Alice is required to send is well-known (see [25, 23, 22]).

▶ **Lemma 7** (Indexing Lower Bound [23]). *Every one-way communication protocol (even with shared randomness) for $\textsc{Index}(N)$ must use $\Omega(N)$ bits of communication.*

3 Single Color Fault

In this section, we study the connectivity problem under one color fault. That is, given two vertices u, v and a faulty color c, one should be able to determine if u, v are connected in $G - c$. In Sections 3.1 and 3.2 we focus on labeling schemes, and provide universally optimal upper and lower bounds. Section 3.3 then discusses routing in the presence of a single forbidden color. In Section 3.5 we change gears and provide centralized oracles for the problem.

3.1 Our Labeling Scheme and the Ball Packing Number

We first show a scheme that works when G is connected. Connectivity cannot be assumed without losing generality, because of the color labels: A color gets *only one* label, which should support connectivity queries in *every* connected component of the input. Later, in Appendix A, we show how to remove this assumption. Consider the following procedure: starting from an arbitrary vertex a_0, iteratively choose a vertex a_i which satisfies

$$\mathrm{dist}_G(a_i, \{a_0, \ldots, a_{i-1}\}) = i,$$

until no such vertex exists. Suppose the procedure halts at the k-th iteration, with the set of chosen vertices $A = \{a_0, \ldots, a_{k-1}\}$. Then every vertex $v \in V$ has distance less than k from A. We use A to construct $O(k \log n)$-bit labels, as follows.

- **Label $L(c)$ of color $c \in [C]$:** For every $a \in A$, store $\mathsf{cid}(a, G - c)$.
- **Label $L(v)$ of vertex $v \in V$:** Let $P(v)$ be a shortest path connecting v to A, and let $a(v)$ be its endpoint in A. For every color c present in $P(v)$, store $\mathsf{cid}(v, G - c)$. Also, store $\mathsf{id}(a(v))$.

Answering queries is straightforward as given $L(v)$ and $L(c)$, one can readily compute $\mathsf{cid}(v, G - c)$: If the color c appears on the path $P(v)$, then $\mathsf{cid}(v, G - c)$ is found in $L(v)$. Otherwise, $P(v)$ connects between v and $a(v)$ in $G - c$, hence $\mathsf{cid}(v, G - c) = \mathsf{cid}(a(v), G - c)$, and the latter is stored in $L(c)$.

The labels have length of $O(\sqrt{n}\log n)$ bits, as follows. Consider the A-vertices chosen at iteration $\lceil k/2 \rceil$ or later. By construction, each of these $\lfloor k/2 \rfloor$ vertices is at distance at least $\lceil k/2 \rceil$ from all others. Hence, the balls of radius $\lfloor k/4 \rfloor$ (in the metric induced by G) centered at these vertices are disjoint, and each such ball contains at least $\lfloor k/4 \rfloor$ vertices. Thus, $\lfloor k/2 \rfloor \cdot \lfloor k/4 \rfloor \le n$, so $k = O(\sqrt{n})$.

The length of the labels assigned by this simple scheme turns out to be not only *existentially optimal*, but also *universally optimal* (both up to a factor of $\log n$). By existential optimality, we mean that every labeling scheme for connectivity under one color fault must have $\Omega(\sqrt{n})$-bit labels on some *worst-case colored graph* G. The stronger universal optimality means that for *every graph topology* G, every such labeling scheme, even tailor-made for G, must assign $\Omega(k)$-bit labels (for some coloring of G).

The Ball-Packing Number. To prove the aforementioned universal optimality of our scheme, we introduce a graph parameter called the *ball-packing number*. As the name suggests, this parameter concerns packing disjoint balls in the metric induced by the graph topology G. Its relation to faulty-color connectivity is hinted by the previous analysis using a "ball packing argument" to obtain the $\tilde{O}(\sqrt{n})$ bound. We next give the formal definitions and some immediate observations.

▶ **Definition 8** (Proper r-ball). *For every integer $r \ge 0$, the* r-ball *in G centered at $v \in V(G)$, denoted $B_G(v, r)$, consists of all vertices of distance at most r from v. That is,*

$$B_G(v, r) = \{u \in V(G) \mid \mathrm{dist}_G(v, u) \le r\}.$$

The r-ball $B_G(v, r)$ is called proper *if there exists $u \in B_G(v, r)$ that realizes the radius, i.e.,* $\mathrm{dist}_G(u, v) = r$*. Note that if the radius r from v is not realized, then there exists $r' < r$ such that the radius r'* is *realized, and $B_G(v, r') = B_G(v, r)$* as sets of vertices*. So, whether $B(v, r)$ is proper depends not only on the set of vertices in this ball, but also on the specified parameter r.*

▶ **Observation 9.** *If $r \le \mathrm{dist}_G(u, v) < \infty$, then $B_G(u, r)$ and $B_G(v, r)$ are proper r-balls.*

▶ **Definition 10** (Ball-packing number). *The* ball-packing number *of G, denoted* $\mathsf{bp}(G)$*, is the maximum integer r such that there exist at least r vertex-disjoint proper r-balls in G.*

▶ **Observation 11.** *(i) For every n-vertex graph G,* $\mathsf{bp}(G) \le \sqrt{n}$*. (ii) For some graphs G, we also have* $\mathsf{bp}(G) = \Omega(\sqrt{n})$ *(e.g., when G is a path).*

A Ball-Packing Upper Bound. Our length analysis for the above scheme in fact showed the existence of at least $\lfloor k/2 \rfloor$ disjoint and proper $\lfloor k/4 \rfloor$-balls, implying that $k = O(\mathsf{bp}(G))$ by Definition 10. Minor adaptations to this scheme to handle several connected components in G yields the following theorem, whose proof is deferred to Appendix A.

▶ **Theorem 12.** *There is a deterministic labeling scheme for connectivity under one color fault that, when given as input an n-vertex graph G, assigns labels of length $O(\mathsf{bp}(G)\log n)$ bits. The query time is $O(1)$ (in the RAM model).*

▶ Remark 13. By Observation 11(i), the label length is always bounded by $O(\sqrt{n}\log n)$ bits.

3.2 A Ball-Packing Lower Bound

We now show an $\Omega(\mathsf{bp}(G))$ bound on the maximal label length.

▶ **Theorem 14.** *Let G be a graph topology. Suppose there is a (possibly randomized) labeling scheme for connectivity under one color fault, that assigns labels of length at most b bits for every coloring of G. Then $b = \Omega(\mathsf{bp}(G))$.*

▶ Remark 15. By the above theorem and Observation 11(ii), every labeling scheme for all topologies must assign $\Omega(\sqrt{n})$-bit labels on some input, which proves Theorem 4 for the special case $f = 1$.

Proof of Theorem 14. Denote $r = \mathsf{bp}(G)$. The proof uses the labeling scheme and the graph topology G to construct a communication protocol for $\textsc{Index}(r^2)$. Let $x = x_0 x_1 \cdots x_{r^2-1}$ be the input string given to Alice, where each $x_i \in \{0,1\}$. Let i^* be the index given to Bob, where $0 \le i^* \le r^2 - 1$. On a high level, the communication protocol works as follows. Both Alice and Bob know the (uncolored) graph topology G in advance, as part of the protocol. Alice colors the edges of her copy of G according to her input x, and applies the labeling scheme to compute labels for the vertices and colors. She then sends $O(r)$ such labels to Bob, and he recovers x_{i^*} by using the labels to answer a connectivity query in the colored graph. As the total number of sent bits is $O(b \cdot r)$, it follows by Lemma 7 that $b \cdot r = \Omega(r^2)$, and hence $b = \Omega(r) = \Omega(\mathsf{bp}(G))$. The rest of this proof is devoted to the full description of the protocol.

In order to color G, Alice does the following. She uses the color palette $\{0, 1, \dots, r-1\} \cup \{\bot\}$, where the symbol $\bot$ is used instead of r to stress that $\bot$ is a special *never failing color* in the protocol. Let $v_0, v_1 \dots, v_{r-1}$ be centers of r disjoint proper r-balls in G, which exist by Definition 10 of Ball-Packing, and since $r = \mathsf{bp}(G)$. For every $k, l \in [0, r)$, define

$$E_{k,l} \overset{\text{def}}{=} \big\{\{u,w\} \in E \mid \text{dist}_G(v_k, u) = l \text{ and } \text{dist}_G(v_k, w) = l+1\big\}.$$

In other words, $E_{k,l}$ is the set of edges connecting layers l and $l+1$ of the k-th ball $B(v_k, r)$. As the layers in a ball are disjoint, and the balls themselves are disjoint, the sets $\{E_{k,l}\}_{k,l}$ are mutually disjoint. Alice colors these edge-sets by the following rule: For every $i \in [0, r^2-1]$, she decomposes it as $i = kr + l$ with $l, k \in [0, r)$. If $x_i = 1$, the edges in $E_{k,l}$ get the color l. Otherwise, when $x_i = 0$, these edges get the null-color $\bot$. Every additional edge in G, outside of the sets $\{E_{k,l}\}_{k,l}$, is also colored by $\bot$. The purpose of this coloring is to ensure the following property, for $k, l \in [0, r)$ and $i = kr + l$: If $x_i = 0$, then (the induced graph on) $B_G(v_k, r)$ does not contain any l-colored edges and its vertices are connected in $G - l$. However, if $x_i = 1$, then $E_{k,l}$ is colored by l, hence in $G - l$, v_k is disconnected from every u for which $\text{dist}_G(u, v_k) > l$.

Next, we describe the message sent by Alice. For $0 \le k \le r-1$, let $u_k \in V$ with $\text{dist}_G(u_k, v_k) = r$, which exists by Definition 8, as $B_G(v_k, r)$ is a *proper* r-ball. Alice applies the labeling scheme on the colored G, and sends to Bob the labels of the vertices $v_0, \dots, v_{r-1}, u_0, \dots, u_{r-1}$, and of the colors $0, \dots, r-1$. This amounts to $3r$ labels.

Finally, we describe Bob's strategy. He decomposes i^* as $i^* = k^* r + l^*$ with $k^*, l^* \in [0, r)$, and uses the labels of v_{k^*}, u_{k^*}, l^* to query the connectivity of v_{k^*} and u_{k^*} in $G - l^*$. If the answer is *disconnected*, Bob determines that $x_{i^*} = 1$, and if it is *connected*, he determines that $x_{i^*} = 0$. By the previously described property of the coloring, Bob indeed recovers x_{i^*} correctly. Thus, this protocol solves $\textsc{Index}(r^2)$, which concludes the proof.

This proof extends quite easily to *vertex-colored* graphs; Alice can color the vertices in the l-th layer of $B(v_k, r)$ instead of the edges $E_{k,l}$. ◀

3.3 Forbidden Color Routing

We next consider designing routing schemes with a forbidden color, with our goal being to prove Theorem 2 (see Section 1.1.1 for definitions and statement).

For the sake of simplicity, we assume that when c is the color to be avoided, the graph $G - c$ is connected. (In particular, this also implies that G is connected.) Intuitively, this assumption is reasonable as we cannot route between different connected components of $G - c$. To check if the routing is even possible (i.e., if s and t are in the same connected component), we can use the connectivity labels of Theorem 12 at the beginning of the procedure. Technically, this assumption can be easily removed, at the cost of introducing some additional clutter.

3.3.1 Basic Tools

We start with some basic building blocks on which our scheme is used. First, we crucially use the existence of the set A constructed in the labeling procedure of Section 3.1. The following lemma summarizes its critical properties:

▶ **Lemma 16.** *There is a vertex set $A \subseteq V$ such that $|A| = O(\mathsf{bp}(G))$, and every vertex $v \in V$ has $\mathrm{dist}_G(v, A) = O(\mathsf{bp}(G))$.*

Next, we use (in a black-box manner) a standard building block in many routing schemes: the *Thorup-Zwick tree routing scheme* [35]. Its properties are summarized in the following lemma:

▶ **Lemma 17** (Tree Routing [35])**.** *Let T be an n-vertex tree. One can assign each vertex $v \in V(T)$ a* routing table *$R_T(v)$ and a* destination label *$L_T(v)$ with respect to the tree T, both of $O(\log n)$ bits. For any two vertices $u, v \in V(T)$, given $R_T(u)$ and $L_T(v)$, one can find the port number of the T-edge from u that heads in the direction of v in T.*

We now define several trees that are crucial for our scheme.

First, we construct a specific spanning tree T of G, designed so that the V-to-A shortest paths in G are tree paths in T. Recall that for every $v \in V$, $P(v)$ is a shortest path connecting v to A, and $a(v)$ is the A-endpoint of this path (see the beginning of Section 3.1). We choose the paths $P(v)$ consistently, so that if vertex u appears on $P(v)$, then $P(u)$ is a subpath of $P(v)$. This ensures that the union of the paths $\bigcup_{v \in V} P(v)$ is a forest. The tree T is created by connecting the parts of this forest by arbitrary edges. We root T at an arbitrary vertex r.

After the failure of color c, the tree T breaks into *fragments* (the connected components of $T - c$). We define the *recovery tree* of color c, denoted T_c, as a spanning tree of $G - c$ obtained by connecting the fragments of $T - c$ via additional edges of $G - c$. These edges are called the *recovery edges* of T_c, and the fragments of $T - c$ are also called fragments of T_c.

For $u, v \in V$ and color c, we denote $e(u, v, c)$ as the first recovery edge appearing in the u-to-v path in T_c (when such exists). Note that we treat this path as directed from u to v. Accordingly, we think of $e(u, v, c)$ as a *directed* edge (x, y) where its first vertex x is closer to u, and its second vertex y is closer to v. Thus, $e(u, v, c)$ and $e(v, u, c)$ may refer to the same edge, but in opposite directions. We will use a basic data block denoted $\mathsf{FirstRecEdge}(u, v, c)$ storing the following information regarding $e(u, v, c)$:

- The port number of $e(u, v, c)$, from its first vertex x to its second vertex y.
- The tree-routing label w.r.t. T of the first vertex x, i.e. $L_T(x)$.
- A Boolean indicating whether the second vertex y and v lie in the same fragment of $T - c$.

Note that $\mathsf{FirstRecEdge}(u, v, c)$ consists of $O(\log n)$ bits.

Finally, we classify the fragments of a recovery tree T_c (and of $T - c$) into two types:

- ***A*-fragments**: fragments that contain at least one vertex from A.
- ***B*-fragments**: fragments that are disjoint from A.

Our construction of T ensures the following property:

▶ **Lemma 18.** *For every color c, if vertex $v \in V$ is in a B-fragment of $T - c$, then $c \in P(v)$, i.e., the color c appears on the path $P(v)$.*

Proof. By construction, the path $P(v)$ is a tree path in T connecting v to some $a \in A$. As v is in a B-fragment of $T - c$, this path cannot survive in $T - c$, hence c appears on it. ◀

3.4 Construction of Routing Tables and Labels

We now formally describe the construction of the tables and labels of our scheme, by Algorithms 1–3. An overview of how these are used to route messages, which provides the intuition behind their construction, is provided in the next Section 3.4.1. At first read, it may be beneficial to skip ahead and start with the overview, while referring to the current section to see how the information storage described there is realized formally.

Algorithm 1 Creating the table $R(v)$ of vertex v.

```
store $R_T(v)$
store port number of the edge from $v$ to its parent in $T$
$c(v) \leftarrow$ color of edge from $v$ to its parent in $T$        ▷ undefined if $v = r$
store $c(v)$
for each vertex $a \in A$ do
    store $\mathsf{FirstRecEdge}(v, a, c(v))$
for each color $c \in P(v)$ do
    store $R_{T_c}(v)$
```

Algorithm 2 Creating the label $L(v)$ of vertex v.

```
store $L_T(v)$
store $a(v)$, the $A$-endpoint of $P(v)$
for each color $c \in P(v)$ do
    $a(v, c) \leftarrow$ an $A$-vertex in the nearest $A$-fragment to $v$ in $T_c$
    store $\mathsf{FirstRecEdge}(a(v, c), v, c)$
    store $L_{T_c}(v)$
```

Algorithm 3 Creating the label $L(c)$ of color c.

```
for each vertex $a \in A$ do
    store $\mathsf{FirstRecEdge}(r, a, c)$
```

Size Analysis. It is easily verified that each **store** instruction in Algorithms 1–3 adds $O(\log n)$ bits of storage. In all of these algorithms, the number of such instructions is $O(|P(v)| + |A|)$, which is $O(\mathsf{bp}(G))$ by Lemma 16. Hence, the total size of any $R(v)$, $L(v)$ or $L(c)$ is $O(\mathsf{bp}(G) \log n)$ bits.

3.4.1 Overview of the Routing Scheme

We are now ready to present our forbidden color routing scheme. Here, we give an overview which conveys the main technical ideas. The formal details are provided in Appendix B. Our scheme is best described via two special cases; in the first case, t is in an A-fragment, and in the second case, the s-to-t path in T_c is only via B-fragments. These two cases are combined to obtain the full routing scheme, essentially by first routing to the A-fragment that is nearest to t in T_c, and then routing from that A-fragment to t (crucially, this route does not contain A-fragments).

First Case: t is in an A-fragment. Suppose an even stronger assumption, that we are actually given a vertex $a^* \in A$ that is in the same fragment as t. We will resolve this assumption only at the wrap-up of this section. The general strategy is to try and follow the s-to-t path in the recovery tree T_c. This path is of the form $P_1 \circ e_1 \circ P_2 \circ e_2 \circ \cdots \circ e_\ell \circ P_\ell$, where each P_i is a path in a fragment X_i of $T - c$, and the e_i edges are recovery edges connecting between fragments, so that X_ℓ is the fragment of t in $T - c$. Rather than following this path directly, our goal will be to route from one fragment to the next, through the corresponding recovery edge.

As there are only $O(\mathsf{bp}(G))$ A-fragments, every $v \in V$ can store $O(\log n)$ bits for each A-fragment. However, the A-fragments depend on the failing color, so the routing table of v cannot store said information for every color. To overcome this obstacle, note that in every fragment in $T - c$ (besides the one containing the root r), the root of the fragment is connected to its parent via a c-colored edge. We leverage this property, and let the root v of every fragment in $T - c$ store, for every $a \in A$, the first recovery edge $e(v, a, c)$ on the path from v to a in T_c. Thus, when reaching the fragment X_i of $T - c$, we first go up as far as possible, until we hit the root of X_i. In the general case, this is a vertex v_i such that the edge to its parent is of color c. Therefore, v_i stores in its table the next recovery edge $e_i = e(v_i, a^*, c)$ we aim to traverse. The special case of $v_i = r$ is resolved using the color labels. The color c stores, for every $a \in A$, the first recovery edge $e(r, a, c)$; at the start of the routing procedure, s extracts the information regarding $e(r, a^*, c)$ and writes it in the header.

So, we discover e_i in v_i, and next we use the Thorup-Zwick routing of Lemma 17 on T to get to the first endpoint of e_i. The path leading us to this endpoint is fault-free (it is contained in the fragment X_i). Then, we traverse e_i, and continue in the same manner in the next fragment X_{i+1}.

Once we reach the A-fragment that contains a^* and t, we again use the Thorup-Zwick routing of Lemma 17 on T. For that we also need $L_T(t)$, which s can learn from the label of t and write in the header at the beginning of the procedure.

Second Case: the s-to-t path in T_c is only via B-fragments. As every vertex v in the s-to-t path in T_c is in a B-fragment of $T - c$, by Lemma 18, $c \in P(v)$. Thus, v can store the relevant tree-routing table $R_{T_c}(v)$. Essentially, every v has to store such routing table for every color in $P(v)$. Also, since $c \in P(v)$, t can store in its label $L(t)$ the tree-routing label $L_{T_c}(t)$, and the latter can be extracted by s and placed on the header of the message at the beginning of the procedure. Hence, we can simply route the message using Thorup-Zwick routing scheme of Lemma 17 on T_c.

Putting It Together. We now wrap-up the full routing procedure. If $c \notin P(t)$, then t is connected to $a(t)$, and we get the first case with $a^* = a(t)$, which can be stored in t's label. Thus, suppose $c \in P(t)$. Since $|P(t)| = O(\mathsf{bp}(G))$, the label of t can store $O(\log n)$ bits for every color on $P(t)$, and specifically for the color c of interest. If t is in an A-fragment in $T-c$, then t can pick an arbitrary A-vertex in its fragment as a^*, and again we reduce to the first case. Suppose t is in a B-fragment in $T-c$. In this case, t sets a^* to be an A-vertex from the nearest A-fragment to t in T_c. The label of t can store a^* and the first recovery edge from a^* towards t (i.e., $e(a^*,t,c)$) At the beginning of the procedure, s can find the information regarding a^* and $e(a^*,t,c)$ in t's label, and write it on the message header. Now, routing from s to the fragment of a^* is by done by the first case, traversing this fragment towards $e(a^*,t,c)$ is done using Thorup-Zwick tree-routing on T, and after taking this edge, we can route the message to t according to the second case.

3.5 Centralized Oracles and Nearest Colored Ancestors

In the *centralized* setting of oracles for connectivity under one color fault, the objective is to preprocess the colored graph G into a low-space centralized data structure (oracle) that, when queried with (the names/ids of) two vertices $u,v\in V$ and a color c, can quickly report if u and v are connected in $G-c$. The labeling scheme of Theorem 12 implies such a data structure with $O(n^{1.5})$ space and $O(1)$ query time.[5] (The bounds for centralized data structures are in the standard RAM model with $\Theta(\log n)$-bit words.) By the lower bound of Theorem 14, such a data structure with space $o(n^{1.5})$ cannot be "evenly distributed" into labels.

However, utilizing centralization, we can achieve $O(n)$ space with only $O(\log\log n)$ query time. This is obtained by a reduction to the *nearest colored ancestor* problem, studied by Muthukrishnan and Müller [28] and by Gawrychowski, Landau, Mozes and Weimann [15]. They showed that a rooted n-vertex forest with colored vertices can be processed into an $O(n)$-space data structure, that given a vertex v and a color c, returns the nearest c-colored ancestor of v (or reports that none exist) in $O(\log\log n)$ time. The reduction is as follows. Choose a maximal spanning forest T for G, and root each tree of the forest in the vertex with minimum id. For each vertex $u\in V$, assign it with the color d of the edge connecting u to its parent in T. Additionally, store $\mathsf{cid}(u,G-d)$ in the vertex u. (The roots get a null-color and store their ids, which are also their cids in every subgraph of G.) Now, construct a nearest color ancestor data structure for T as in [28, 15]. Given a query $v\in V$ and color c, we can find the nearest c-colored ancestor w of v in $O(\log\log n)$ time. As w is nearest, the T-path from v to w in T does not contain c-colored edges, implying that $\mathsf{cid}(v,G-c)=\mathsf{cid}(w,G-c)$, and the latter is stored at w. (If no such c-colored ancestor exists, take w as the root, and proceed similarly.) Given $u,v\in V$ and color c, apply the above procedure twice, and determine the connectivity of u,v in $G-c$ by comparing their cids, within $O(\log\log n)$ time. We therefore get:

▶ **Theorem 19.** *Every colored n-vertex graph G can be processed into an $O(n)$-space centralized oracle that given a query of $u,v\in V$ and color c, reports if u,v are connected in $G-c$ in $O(\log\log n)$ time.*

The reduction raises an alternative approach for constructing connectivity labels for one color fault, via providing a labeling scheme for the nearest colored ancestor problem. In Appendix C we show that indeed, such a scheme with $\tilde{O}(\sqrt{n})$-bit labels exists.

[5] The data structure stores all vertex labels, and the labels of all colors that appear in some fixed maximal spanning forest T of G. We can ignore all other colors, as their failure does not change the connectivity in G.

4 Reduction from All-Pairs to Single-Source

In the *single-source* variant of fault-tolerant connectivity, the input graph G comes with a designated *source vertex* s. The queries to be supported are of the form $\langle u, F\rangle$, where $u \in V$ and F is a faulty set of size at most f. It is required to report if u is connected to the source s in $G - F$. The following result shows that this variant is equivalent to the all-pairs variant, up to $\log n$ factors. The result holds whether the faults are edges, vertices, or colors, hence we do not specify the type of faults.

▶ **Theorem 20.** *Let $f \geq 1$. Suppose there is a (possibly randomized) single-source f fault-tolerant connectivity labeling scheme that assigns labels of at most $b(n, f)$ bits on every n-vertex graph. Then, there is a randomized all-pairs f fault-tolerant connectivity labeling scheme that assigns labels of length $O(b(n+1, f) \cdot \log^2 n)$ bits on every n-vertex graph.*

Proof.

Labeling. For each i, j with $1 \leq i \leq \lceil \alpha \ln(n)/\ln(0.9)\rceil$, $1 \leq j \leq \lceil \log_2 n\rceil + 2$ we independently construct a graph G_{ij} as follows: Start with G, add a new vertex s_{ij}, and independently for each $v \in V$, add a new edge connecting s_{ij} to v with probability 2^{-j}. The vertex s_{ij} and the new edges are treated as *non-failing*. That is, in case of color faults, they get a null-color $\perp$ that does not appear in G. For each element (vertex/edge/color) x of G, its label $L(x)$ is the concatenation of all $L_{ij}(x)$, where the $L_{ij}(\cdot)$ are the labels given by the single-source scheme to the instance G_{ij} with designated source s_{ij}. The claimed length bound is immediate.

Answering queries. Let $u, w \in V$, and let F be a fault-set of size at most f. Given $L(u), L(w)$ and $\{L(x) \mid x \in F\}$, we should determine if u, w are connected in $G - F$. To this end, for each i, j, we use the $L_{ij}(\cdot)$ labels of u, F to determine if u is connected to s_{ij} in G_{ij}, and do the same with w instead of u. If the answers are always identical for u and w, we output *connected*. Otherwise, we output *disconnected*.

Analysis. We have made only $O(\log^2 n)$ queries using the single-source scheme, so, with high probability, all of these are answered correctly. Assume this from now on.

If u, w are connected in $G - F$, then this is also true for all $G_{ij} - F$, so they must agree on the connectivity to s_{ij} in this graph. Hence, in this case, the answers for u and w are always identical, and we correctly output *connected*.

Suppose now that u and w are disconnected in $G - F$. Let U be the set of vertices in u's connected component in $G - F$. Define W analogously for w. Without loss of generality, assume $|U| \leq |W|$. Let j be such that $2^{j-2} < |U| \leq 2^{j-1}$. Let $N_U^{(i)}$ be the number of edges between s_{ij} and U in G_{ij}, and define $N_W^{(i)}$ similarly. By Markov's inequality,

$$\Pr\left[N_U^{(i)} = 0\right] \geq 1 - \mathbb{E}\left[N_U^{(i)}\right] = 1 - |U| \cdot 2^{-j} \geq 1 - 2^{j-1} \cdot 2^{-j} = 1/2.$$

On the other hand,

$$\Pr[N_W^{(i)} \geq 1] = 1 - \left(1 - 2^{-j}\right)^{|W|} \geq 1 - \left(1 - 2^{-j}\right)^{2^{j-2}} \geq 1 - e^{-1/4} > 0.2.$$

Since U and W are disjoint, $N_U^{(i)}$ and $N_W^{(i)}$ are independent random variables. Hence, with probability at least 0.1, the source s_{ij} is connected to w but not to u in $G_{ij} - F$, and the answers for u and w given by the $L_{ij}(\cdot)$-labels are different. As the graphs $\{G_{ij}\}_i$ are formed independently, the probability there exists an i for which w is connected to s_{ij} and u is disconnected from s_{ij} is at least $1 - (0.9)^{\alpha \ln n/\ln(0.9)} = 1 - 1/n^{\alpha}$. In this case, the output is *disconnected*, as required. ◀

References

1 Ittai Abraham, Shiri Chechik, and Cyril Gavoille. Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC*, pages 1199–1218. ACM, 2012. doi:10.1145/2213977.2214084.

2 Ittai Abraham, Shiri Chechik, Cyril Gavoille, and David Peleg. Forbidden-set distance labels for graphs of bounded doubling dimension. *ACM Trans. Algorithms*, 12(2):22:1–22:17, 2016. doi:10.1145/2818694.

3 Surender Baswana, Koustav Bhanja, and Abhyuday Pandey. Minimum+1 (s, t)-cuts and dual edge sensitivity oracle. In *49th International Colloquium on Automata, Languages, and Programming, ICALP*, volume 229 of *LIPIcs*, pages 15:1–15:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICALP.2022.15.

4 Greg Bodwin, Fabrizio Grandoni, Merav Parter, and Virginia Vassilevska Williams. Preserving distances in very faulty graphs. In *44th International Colloquium on Automata, Languages, and Programming, ICALP*, volume 80 of *LIPIcs*, pages 73:1–73:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.73.

5 Shiri Chechik. Improved distance oracles and spanners for vertex-labeled graphs. In *Algorithms - ESA 2012 - 20th Annual European Symposium*, volume 7501 of *Lecture Notes in Computer Science*, pages 325–336. Springer, 2012. doi:10.1007/978-3-642-33090-2_29.

6 Keerti Choudhary. An optimal dual fault tolerant reachability oracle. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP*, volume 55 of *LIPIcs*, pages 130:1–130:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.130.

7 David Coudert, Pallab Datta, Stephane Perennes, Hervé Rivano, and Marie-Emilie Voge. Shared risk resource group complexity and approximability issues. *Parallel Process. Lett.*, 17(2):169–184, 2007. doi:10.1142/S0129626407002958.

8 Bruno Courcelle, Cyril Gavoille, Mamadou Moustapha Kanté, and Andrew Twigg. Connectivity check in 3-connected planar graphs with obstacles. *Electron. Notes Discret. Math.*, 31:151–155, 2008. doi:10.1016/j.endm.2008.06.030.

9 Bruno Courcelle and Andrew Twigg. Compact forbidden-set routing. In *Proceedings 24th Annual Symposium on Theoretical Aspects of Computer Science, STACS*, volume 4393 of *Lecture Notes in Computer Science*, pages 37–48. Springer, 2007. doi:10.1007/978-3-540-70918-3_4.

10 Michal Dory and Merav Parter. Fault-tolerant labeling and compact routing schemes. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 445–455, 2021. doi:10.1145/3465084.3467929.

11 Ran Duan and Seth Pettie. Dual-failure distance and connectivity oracles. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 506–515, 2009. doi:10.1137/1.9781611973068.56.

12 Georgios Ellinas, Eric Bouillet, Ramu Ramamurthy, J.-F Labourdette, Sid Chaudhuri, and Krishna Bala. Routing and restoration architectures in mesh optical networks. *Opt Networks Mag*, 4, January 2003.

13 Jacob Evald, Viktor Fredslund-Hansen, and Christian Wulff-Nilsen. Near-optimal distance oracles for vertex-labeled planar graphs. In *32nd International Symposium on Algorithms and Computation, ISAAC*, volume 212 of *LIPIcs*, pages 23:1–23:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ISAAC.2021.23.

14 Kyle Fox, Debmalya Panigrahi, and Fred Zhang. Minimum cut and minimum k-cut in hypergraphs via branching contractions. *ACM Trans. Algorithms*, 19(2):13:1–13:22, 2023. doi:10.1145/3570162.

15 Pawel Gawrychowski, Gad M. Landau, Shay Mozes, and Oren Weimann. The nearest colored node in a tree. *Theor. Comput. Sci.*, 710:66–73, 2018. doi:10.1016/j.tcs.2017.08.021.

16 Mohsen Ghaffari, David R. Karger, and Debmalya Panigrahi. Random contractions and sampling for hypergraph and hedge connectivity. In *Proceedings of the Twenty-Eighth Annual*

ACM-SIAM Symposium on Discrete Algorithms, SODA, pages 1101–1114, 2017. doi:10.1137/1.9781611974782.71.

17 Manoj Gupta and Shahbaz Khan. Multiple source dual fault tolerant BFS trees. In *44th International Colloquium on Automata, Languages, and Programming, ICALP*, volume 80 of *LIPIcs*, pages 127:1–127:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.127.

18 Bernhard Haeupler, David Wajc, and Goran Zuzic. Universally-optimal distributed algorithms for known topologies. In *53rd Annual ACM SIGACT Symposium on Theory of Computing, STOC*, pages 1166–1179, 2021. doi:10.1145/3406325.3451081.

19 Danny Hermelin, Avivit Levy, Oren Weimann, and Raphael Yuster. Distance oracles for vertex-labeled graphs. In *Automata, Languages and Programming - 38th International Colloquium, ICALP*, volume 6756 of *Lecture Notes in Computer Science*, pages 490–501. Springer, 2011. doi:10.1007/978-3-642-22012-8_39.

20 Jian Qiang Hu. Diverse routing in optical mesh networks. *IEEE Transactions on Communications*, 51(3):489–494, 2003. doi:10.1109/TCOMM.2003.809779.

21 Taisuke Izumi, Yuval Emek, Tadashi Wadayama, and Toshimitsu Masuzawa. Deterministic fault-tolerant connectivity labeling scheme. In *Symposium on Principles of Distributed Computing, PODC*, pages 190–199. ACM, 2023. doi:10.1145/3583668.3594584.

22 T. S. Jayram, Ravi Kumar, and D. Sivakumar. The one-way communication complexity of hamming distance. *Theory Comput.*, 4(1):129–135, 2008. doi:10.4086/toc.2008.v004a006.

23 Ilan Kremer, Noam Nisan, and Dana Ron. On randomized one-round communication complexity. *Comput. Complex.*, 8(1):21–49, 1999. doi:10.1007/s000370050018.

24 F. Kuipers. An overview of algorithms for network survivability. *ISRN Communications and Networking*, 2012, December 2012. doi:10.5402/2012/932456.

25 Eyal Kushilevitz and Noam Nisan. *Communication Complexity.* Cambridge University Press, 1996. doi:10.1017/CBO9780511574948.

26 Itay Laish and Shay Mozes. Efficient dynamic approximate distance oracles for vertex-labeled planar graphs. *Theory Comput. Syst.*, 63(8):1849–1874, 2019. doi:10.1007/s00224-019-09949-5.

27 Eytan H. Modiano and Aradhana Narula-Tam. Survivable lightpath routing: a new approach to the design of wdm-based networks. *IEEE J. Sel. Areas Commun.*, 20(4):800–809, 2002. doi:10.1109/JSAC.2002.1003045.

28 S. Muthukrishnan and Martin Müller. Time and space efficient method-lookup for object-oriented programs. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 42–51, 1996. URL: http://dl.acm.org/citation.cfm?id=313852.313882.

29 Merav Parter. Dual failure resilient BFS structure. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC*, pages 481–490, 2015. doi:10.1145/2767386.2767408.

30 Merav Parter. Distributed constructions of dual-failure fault-tolerant distance preservers. In *34th International Symposium on Distributed Computing, DISC*, volume 179 of *LIPIcs*, pages 21:1–21:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.21.

31 Merav Parter and David Peleg. Sparse fault-tolerant BFS trees. In *Algorithms - ESA 2013 - 21st Annual European Symposium*, volume 8125 of *Lecture Notes in Computer Science*, pages 779–790. Springer, 2013. doi:10.1007/978-3-642-40450-4_66.

32 Merav Parter and Asaf Petruschka. Õptimal dual vertex failure connectivity labels. In *36th International Symposium on Distributed Computing, DISC*, volume 246 of *LIPIcs*, pages 32:1–32:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.DISC.2022.32.

33 Merav Parter, Asaf Petruschka, and Seth Pettie. Connectivity labeling and routing with multiple vertex failures. In *Proceedings of the 56th Annual ACM Symposium on Theory of*

Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024, pages 823–834. ACM, 2024. `doi:10.1145/3618260.3649729`.

34 Asaf Petruschka, Shay Sapir, and Elad Tzalik. Color fault-tolerant spanners. In *15th Innovations in Theoretical Computer Science Conference, ITCS*, volume 287 of *LIPIcs*, pages 88:1–88:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. `doi:10.4230/LIPICS.ITCS.2024.88`.

35 Mikkel Thorup and Uri Zwick. Compact routing schemes. In *Proceedings 13th ACM Symposium on Parallel Algorithms and Architectures, SPAA*, pages 1–10, 2001. `doi:10.1145/378580.378581`.

36 Dekel Tsur. Succinct data structures for nearest colored node in a tree. *Inf. Process. Lett.*, 132:6–10, 2018. `doi:10.1016/j.ipl.2017.10.001`.

37 Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977. `doi:10.1007/BF01683268`.

38 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983. `doi:10.1016/0020-0190(83)90075-3`.

39 Rupei Xu and Warren Shull. Hedge connectivity without hedge overlaps. *CoRR*, 2020. `arXiv:2012.10600`.

40 Peng Zhang, Jin-Yi Cai, Lin-Qing Tang, and Wen-Bo Zhao. Approximation and hardness results for label cut and related problems. *Journal of Combinatorial Optimization*, 21:192–208, 2011. `doi:10.1007/s10878-009-9222-0`.

A Single Color Fault: Proof of Theorem 12

Labeling. The labeling procedure is presented as Algorithm 4.

Algorithm 4 Labeling for one color fault.

```
Require: Colored graph $G$.
Ensure: Labels $L(v)$ for each vertex $v \in V$, and $L(c)$ for each color $c$.
$A_0 \leftarrow \{a \in V \mid \mathsf{id}(a) = \mathsf{cid}(a, G)\}$ ▷ vertices w/ min id in each connected component of $G$
$A \leftarrow \emptyset$
$i \leftarrow 1$
while there exists vertex $a_i \in V$ with $\mathrm{dist}_G(a_i, A_0 \cup A) = i$ do
    $A \leftarrow A \cup \{a_i\}$
    $i \leftarrow i + 1$
for each vertex $v \in V$ do                                   ▷ create the label $L(v)$
    $a(v) \leftarrow$ a closest vertex to $v$ from $A_0 \cup A$ in $G$
    $P(v) \leftarrow$ a shortest $v$-to-$a(v)$ path in $G$
    store in $L(v)$ the id of $a(v)$, $\mathsf{id}(a(v))$
    store in $L(v)$ a dictionary that maps key $d$ to value $\mathsf{cid}(v, G - d)$, for every color $d$
on $P(v)$
for each color $c$ do                                          ▷ create the label $L(c)$
    store in $L(c)$ the name of the color $c$
    store in $L(c)$ a dictionary that maps key $\mathsf{id}(a)$ to value $\mathsf{cid}(a, G - c)$, for each $a \in A$
```

Answering queries. As before, it suffices to show that one can report $\mathsf{cid}(v, G - c)$ merely from the labels $L(v)$ and $L(c)$. This is done as follows. If the key c appears in the dictionary of $L(v)$, then $\mathsf{cid}(v, G-c)$ is the corresponding value, and we are done. Otherwise, $P(v)$ does not contain the color c, so it connects v to $a(v)$ in $G-c$, and therefore $\mathsf{cid}(v, G-c) = \mathsf{cid}(a(v), G-c)$.

Recall that $a(v) \in A_0 \cup A$. If the key $\mathsf{id}(a(v))$ appears in the dictionary of $L(c)$, then $\mathsf{cid}(a(v), G - c)$ is the corresponding value, and we are done. Otherwise, it must be that $a(v) \in A_0$. Recall that A_0 contains vertices having minimum id in their connected component in G. This implies that $\mathsf{cid}(a(v), G - c) = \mathsf{id}(a(v))$, and the latter is stored in $L(v)$.

Length analysis. Let k be the halting iteration of the while loop in Algorithm 4 (line 4). Then $|A| = k - 1$, so the length of each color label is $O(k \log n)$ bits. Also, by the while condition, each vertex $v \in V$ has distance less than k from $A_0 \cup A$ in G, so $P(v)$ contains less than k edges, hence also less than k colors. Therefore, the length of $L(v)$ is also $O(k \log n)$ bits.

We now prove that $k = O(\mathsf{bp}(G))$. Let $A' = \{a_i \mid i \geq \lceil k/2 \rceil\}$ be the "second half" of chosen A-vertices. Note that $|A'| \geq k/2$. For each $i = 1, \ldots k - 1$, denote by A_i the state of set A right after the i-th iteration. If $a_i, a_j \in A'$ with $i > j$, then $a_j \in A_{i-1}$, hence

$$\mathrm{dist}_G(a_i, a_j) \geq \mathrm{dist}_G(a_i, A_0 \cup A_{i-1}) = i > \lceil k/2 \rceil.$$

Therefore, $B(a_i, \lfloor k/4 \rfloor)$ and $B(a_j, \lfloor k/4 \rfloor)$ are disjoint. Also, if $a_i \in A'$, then $\mathrm{dist}_G(a_i, A_0 \cup A) = i > \lfloor k/4 \rfloor$, so Observation 9 implies that $B(a_i, \lfloor k/4 \rfloor)$ is proper. Thus, the collection of $\lfloor k/4 \rfloor$-balls centered at the A' vertices certifies that $\lfloor k/4 \rfloor \leq \mathsf{bp}(G)$, which concludes the proof.

▶ Remark 21. The proof extends seamlessly to vertex-colored graphs. It is worth noting that if the vertex $a(v)$ has the color c, then $\mathsf{cid}(v, G - c)$ is stored in $L(v)$ (since $a(v) \in P(v)$).

B Forbidden Color Routing

This section gives the formal description of the routing procedure, overviewed in Section 3.4.1.

At the beginning of the procedure, s holds the labels $L(s), L(t)$ and $L(c)$ and should route the message M to t avoiding the color c. As described in Section 3.4.1, the routing procedure will have two phases. In the first phase, the message is routed to the fragment of $T - c$ that contains a carefully chosen vertex $a^* \in A$. In the second phase, it is routed from this fragment to the target t.

Initialization at s. First, s determines the vertex a^* as follows: If $c \in P(t)$, then $a^* = a(t, c)$, which is found in $L(t)$. Otherwise, $a^* = a(t)$, the A-endpoint of $P(t)$, which is again found in $L(t)$. Next, s creates the initial header of the message M, that contains:

- The name of the color c.
- The name of the vertex a^*.
- The block $\mathsf{FirstRecEdge}(r, a^*, c)$, found in $L(c)$.
- The tree-routing label $L_T(t)$, found in $L(t)$.
- If $c \in P(t)$, the block $\mathsf{FirstRecEdge}(a^*, t, c)$ and the tree-routing label $L_{T_c}(t)$, found in $L(T)$.

This information will permanently stay in the header of M throughout the routing procedure, and we refer to it as the *permanent header*. Verifying that it requires $O(\log n)$ bits is immediate.

First Phase: Routing from s to the Fragment of a^*. As in Section 3.4.1, let $e_1, e_2, \ldots e_\ell$ be the recovery edges on the s-to-a^* path in T_c (according to order of appearance), each connecting between fragments X_{i-1} and X_i of $T - c$. Denote by v_i the root of fragment X_i, and thus $e_i = e(v_i, a^*, c)$. Recall that we aim to route the message through these edges and reach the last fragment X_ℓ, which is an A-fragment containing a^*, according to the following strategy: Upon reaching a fragment $X_i \neq X_\ell$, we go up until we reach its root v_i, extract information regarding the next recovery edge e_i, and use Thorup-Zwick tree-routing on T in order to get to e_i. We then traverse it to reach X_{i+1}, and repeat the process.

We now describe this routing procedure formally. The header of M contains two updating fields:

- M.UP: stores a Boolean value
- M.NEXT: stores a block referring to the next recovery edge in the path, of the form FirstRecEdge$(\cdot, a^*, c)$.

Clearly, this requires $O(\log n)$ bits of storage. We will maintain the following invariant:

(I): If the message is currently in the fragment $X_i \neq X_\ell$, and M.UP $= 0$, then M.NEXT stores information referring to e_i.

At initialization, s sets M.UP $\leftarrow$ *True* and M.NEXT $\leftarrow \perp$ (a null symbol), which trivially satisfies the invariant (I). We will use the field M.UP also to mark that we are still in the first phase of the routing. During the first phase, it will be a valid Boolean value. When we reach the fragment X_ℓ, we set M.UP to $\perp$ to notify the beginning of the second phase. While M.UP $\neq \perp$ (i.e., during the first phase), upon the arrival of M to a vertex v, it executes the code presented in Algorithm 5 to determine the next hop.

Algorithm 5 First phase: Routing M from v towards the fragment of a^* (while M.UP $\neq \perp$).

```
if $M$.UP $= True$ then
    if $v \neq r$ and $c(v) \neq c$ then
        send $M$ through the port to $v$'s parent in $T$, found in $R(v)$
    else
        find FirstRecEdge$(v, a^*, c)$, in permanent header when $v = r$, or in $R(v)$ when $c(v) = c$
        $M$.NEXT $\leftarrow$ FirstRecEdge$(v, a^*, c)$
        $M$.UP $\leftarrow False$
if $M$.UP $= False$ then
    $x \leftarrow$ first vertex of the edge $e$ found in $M$.NEXT
    if $v \neq x$ then
        send $M$ in direction of $x$ in $T$, using $L_T(x)$ from $M$.NEXT, and $R_T(v)$ from $R(v)$
    else                                                          ▷ $v = x$
        if second vertex of $e$ is in $X_\ell$ (as indicated by $M$.NEXT) then
            $M$.UP $\leftarrow \perp$                 ▷ will reach $X_\ell$ in next step, and then done
        else
            $M$.UP $\leftarrow True$
        send $M$ through the port of the edge $e$ found in $M$.NEXT
```

Invariant (I) is maintained when setting M.UP $\leftarrow$ *False* in Algorithm 5, since we previously set M.NEXT to $e(v, a^*, c)$, which is the first recovery edge in the v-to-a^* path in T_c, and thus, when $v \in X_i$, this edge equals e_i. When the message first reaches a vertex $s' \in X_\ell$, the field M.UP contains a null value $\perp$, and we start the second phase of the routing procedure, as described next.

Second Phase: Routing from the Fragment of a^* to t. Here, we use the careful choice of the vertex a^*. The easier case is when $c \notin P(t)$. In this case, $a^* = a(v)$, and t is in the same fragment as a^* by Lemma 18. Therefore, we can use the Thorup-Zwick routing of

Lemma 17 on T to route the message M from s' to t. Note that the label $L_T(t)$ is found in the permanent header, and that each intermediate vertex v on the path has $R_T(v)$ in its table.

We now treat the case where $c \in P(t)$. In this case, $a^* = a(t, c)$, which is defined to be an A-vertex in the nearest A-fragment to t in T_c, and hence, X_ℓ is that nearest A-fragment to t in T_c. Since $c \in P(t)$, the permanent header stores $\mathsf{FirstRecEdge}(a^*, t, c)$, or specifies that the first recovery edge $e(a^*, t, c)$ is undefined, i.e. that a^* and t are in the same fragment. In the latter case, we can act exactly as above and route the message over T. So, assume $\mathsf{FirstRecEdge}(a^*, t, c)$ is found.

Let x and y be the first and second vertices of $e(a^*, t, c)$. Then $\mathsf{FirstRecEdge}(a^*, t, c)$ contains $L_T(x)$, so we can route the message between $s', x \in X_\ell$ using the Thorup-Zwick routing of Lemma 17 on T. We then traverse the edge $e(a^*, t, c)$ from x to y, where the relevant port is stored in the permanent header. We now make the following important observation:

▶ **Lemma 22.** *If v is a vertex on the y-to-t path in T_c, then its table $R(v)$ must contain $R_{T_c}(v)$.*

Proof. First, we note that v must be a B-fragment. Indeed, if v were in an A-fragment, then this would be a closer A-fragment to t than X_ℓ in T_c, which is a contradiction. Therefore, by Lemma 18, it must be that $c \in P(v)$. This means that $R_{T_c}(v)$ is stored in $R(v)$ by Algorithm 1. ◀

Note that $L_{T_c}(t)$ is found in the permanent header, as $c \in P(t)$. Thus, we can route M from y to t along the connecting path in the recovery tree T_c, using the Thorup-Zwick routing of Lemma 17 on T_c. This concludes the routing procedure.

C Nearest Colored Ancestor Labels

In this section, we show how the $O(n)$-space, $O(\log\log n)$-query time nearest colored ancestor data structure of [15] can be used to obtain $O(\sqrt{n}\log n)$-bit labels for this problem.

The labels version of nearest colored ancestor is formally defined as follows. Given a rooted n-vertex forest T with colored vertices, where each vertex v has an arbitrary unique $O(\log n)$-bit identifier $\mathsf{id}(v)$, the goal is to assign short labels to each vertex and color in T, so that the id of the nearest c-colored ancestor of vertex v can be reported by inspecting the labels of c and v. The reduction from the centralized setting implies that such a labeling scheme can be used “as is”for connectivity under one color fault.[6] First, by Theorem 14, we get:

▶ **Corollary 23.** *Every labeling scheme for nearest colored ancestor in n-vertex forests must have label length $\Omega(\sqrt{n})$ bits. Furthermore, this holds even for paths, as their ball-packing number is $\Omega(\sqrt{n})$.*

▶ Remark 24. The above lower bound can be strengthened to $\Omega(\sqrt{n}\log n)$. This is by considering the problem that our scheme *actually* solves: report the minimum id of a vertex connected to v in $G - c$, from the labels of v and c. For this problem, one can extend the proof of Theorem 14 to show an $\Omega(\sqrt{n}\log n)$-bit lower bound for paths. The reduction described above shows that a nearest colored ancestor labeling scheme can be used to report such minimum ids.

[6] When constructing the nearest colored ancestor labels in the reduction, we augment the id of each vertex v with $\mathsf{cid}(v, G - c)$, where c is the color of the tree edge from v to its parent.

The data structure in [15] can, in fact, be transformed into $O(\sqrt{n}\log n)$-bit labels. We first briefly explain how this data structure works. Each vertex v gets two *time-stamps* $\mathsf{pre}(v), \mathsf{post}(v)$, which are the first and last times a DFS traversal in T reaches v. The time-stamps of c-colored vertices are inserted to a *predecessor structure* [37, 38] for color c. For each time-stamp of a (c-colored) vertex u, we also store the id of the nearest c-colored (strict) ancestor of u. A query (v, c) is answered by finding the predecessor of $\mathsf{pre}(v)$ in the structure of c. If the result is $\mathsf{pre}(u)$, then u is returned. If it is $\mathsf{post}(u)$, then the ancestor pointed by u is returned. Correctness follows by standard properties of DFS time-stamps. The predecessor structure for c answers queries in $O(\log\log n)$ time, and takes up $O(|V_c|)$ space (in words), where V_c is the set of c-colored vertices. The total space is $O(\sum_c |V_c|) = O(n)$.

To construct the labels, let $\mathcal{H} = \{c \mid |V_c| \geq \sqrt{n}\}$ be the *highly prevalent* colors, and $\mathcal{R}$ be the rest of the colors. As there are only n vertices, $|\mathcal{H}| = O(\sqrt{n})$. We can therefore afford to let each vertex v explicitly store in its label the id of its nearest c-color ancestor, for each $c \in \mathcal{H}$. To handle the remaining $\mathcal{R}$-colors, we store in the label of each $c \in \mathcal{R}$ the predecessor structure for c, which only requires $O(|V_c|\log n) = O(\sqrt{n}\log n)$ bits. By augmenting the vertex labels with $\mathsf{pre}(\cdot)$ times (requiring only $O(\log n)$ additional bits), we can also answer queries with colors in $\mathcal{R}$. We obtain:

▶ **Corollary 25.** *There is a labeling scheme for nearest colored ancestor in n-vertex forests, with labels of length $O(\sqrt{n}\log n)$ bits. Queries are answered in $O(\log\log n)$ time.*

Sing a Song of Simplex

Victor Shoup
Offchain Labs, New York City, NY, USA

Abstract

We flesh out some details of the recently proposed Simplex atomic broadcast protocol, and modify it so that leaders disperse blocks in a more communication-efficient fashion. The resulting protocol, called *DispersedSimplex*, maintains the simplicity and excellent – indeed, optimal – latency characteristics of the original Simplex protocol. We also present a variant that supports "stable leaders". We also suggest a number of practical optimizations and provide concrete performance estimates that take into account not just network latency but also network bandwidth limitations and computational costs.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols

Keywords and phrases Consensus, Atomic broadcast, Blockchain

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.37

Related Version *Full Version*: `https://eprint.iacr.org/2023/1916` [22]

Acknowledgements Thanks to Benjamin Chan and Rafael Pass for helpful discussions on the Simplex protocol. Thanks to Ed Felten for suggesting the "packet-switching pipeline" strategy in Section 3.5.

1 Introduction

Byzantine fault tolerance (BFT) is the ability of a computing system to endure arbitrary (i.e., Byzantine) failures of some of its components while still functioning properly as a whole. One approach to achieving BFT is via state machine replication [21]: the logic of the system is replicated across a number of machines, each of which maintains state, and updates its state by executing a sequence of transactions. In order to ensure that the non-faulty machines end up in the same state, they must each deterministically execute the same sequence of transactions. This is achieved by using a protocol for *atomic broadcast*.

In an atomic broadcast protocol, we have a committee of n parties, some of which are honest (and follow the protocol), and some of which are corrupt (and may behave arbitrarily). Roughly speaking, such an atomic broadcast protocol allows the honest parties to schedule a sequence of transactions in a consistent way, so that each honest party schedules the same transactions in the same order. Each party receives various transactions as input – these inputs are received incrementally over time, not all at once. It may be required that a transaction satisfy some type of validity condition, which can be verified locally by each party. These details are application specific and will not be further discussed. Each party outputs an ordered sequence of transactions – these outputs are generated incrementally, not all at once. One key security property of any secure atomic broadcast protocol is **safety**, which means that each honest party outputs the same sequence of transactions. Another key property of any secure atomic broadcast protocol is **liveness**. There are different notions of liveness one can consider, but the basic idea is that the protocol should not get stuck and stop outputting transactions.

Different protocols make different assumptions about the latency guarantees of the network and the number of corrupt parties. Here, we assume that the number of corrupt parties is less than $n/3$, and we consider protocols that are guaranteed to provide safety without any latency assumption, and that are guaranteed to provide liveness only in intervals of "network synchrony", in which the latency is below a certain defined threshold. This is the *partial*

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 37; pp. 37:1–37:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

synchrony model, introduced in [12]. The bound of $n/3$ on the number of corrupt parties is optimal in this model. Many quite practical atomic broadcast protocols have been proposed in this model, starting with the classic PBFT protocol [8], and this is still an area of active research.

In this paper, we consider the recently proposed Simplex atomic broadcast protocol [9]. Like many other recent protocols in this space (such as HotStuff [27] and HotStuff-2 [19]), Simplex is a leader-based, permissioned blockchain protocol: the protocol proceeds in slots (a.k.a., views, rounds), so that in each slot a leader proposes a block of transactions, and these blocks get added to a tree of blocks. Over time, a path of committed blocks in this tree emerges – safety ensures that all parties agree on the same path of committed blocks. In these protocols, leaders typically are rotated in each slot – either in a round-robin fashion or using some pseudo-random sequence – which also has the nice effect of mitigating against censorship of transactions. The protocol relies on authenticated communication links and a PKI to support digital signatures (preferably aggregate or threshold signatures for better communication complexity).

Simplex is a wonderfully simple, efficient, and elegant protocol. In this paper, we add to the Simplex story in a number of ways:

- We flesh out some missing (but crucial) details of the Simplex protocol that are needed to get a protocol with acceptable communication complexity. Along the way, we make a few other simplifications; in particular, we observe that while the Simplex protocol as specified in [9] relies on hash-based chaining of blocks, this turns out to be unnecessary.
- More importantly, we modify the protocol so that leaders disperse blocks in a more communication-efficient fashion, while maintaining its simplicity and excellent – indeed, optimal – latency characteristics. We call this variation on the Simplex protocol **DispersedSimplex**.
- We give a detailed analysis of DispersedSimplex (safety, liveness, and performance), and discuss a number of important implementation details, arguing – based on concrete micro-benchmarks and realistic assumptions on network behavior – that despite its simplicity, in typical scenarios, DispersedSimplex should perform quite well in practice, even for $n \approx 100$.
- We present and analyze a variant of DispersedSimplex that supports "stable leaders" (the paper [9] did not investigate such a variant). We argue that this variant can achieve even better performance, mainly because a stable leader can drive the protocol at a significantly faster rate than a constantly rotating leader. The mechanism for failing over from an unresponsive leader is very simple and lightweight (no more complicated or expensive than rotating leaders as in the basic version of the protocol).

In the full version [22] of this paper, we also show how to improve communication complexity using the improved data dissemination techniques of [16, 17], and (perhaps of more theoretical interest) how to get by without any signatures (at the expense of somewhat higher latency).

In Appendix C, we also compare DispersedSimplex to other protocols in the literature. As we will argue, DispersedSimplex, especially the variants that combine stable leaders and better data dissemination techniques, should perform as well as or better than any other state-of-the-art atomic broadcast protocol (including leader-based protocols such as HotStuff [27] and HotStuff-2 [19], as well as DAG-based protocols such as [23]), at least in terms of common-case throughput and latency. Again, these arguments are based on concrete micro-benchmarks and assumptions on network behavior, and they suggest that it would be worthwhile to measure the actual performance of a well-engineered implementation.

2 The DispersedSimplex protocol

Like many other protocols in this area, the Simplex protocol iterates through slots (a.k.a., views, rounds), where in each slot there is a designated leader who proposes a new block, which is chained to a parent block, and two rounds of voting are used to commit the block. Moreover, to improve latency, the protocol is "pipelined", in the sense that it optimistically moves onto the next slot as soon as the first round of voting succeeds, before the block for that slot is committed. Leaders may be rotated in each slot, either in a round-robin fashion or using some pseudo-random sequence. The DispersedSimplex protocol has the same structure as the Simplex protocol; however, instead of broadcasting the block directly, the slot leader uses well-known techniques for information dispersal to disseminate large blocks in a way that keeps the overall communication complexity low and avoids a bandwidth bottleneck at the leader. In particular, the communication is *balanced*, meaning that each party, including the leader, transmits roughly the same about of data over the network. We will show how the information dispersal can be interleaved with the proposal phase and the first voting round so that no extra latency is incurred.

2.1 Preliminaries

We have a committee of n parties, $P_1, \ldots, P_n$, at most $t < n/3$ of which are corrupt. We assume the parties are connected by authenticated point-to-point channels. We will not generally assume network synchrony. However, we say the network is *δ-synchronous over an interval* $[a, b+\delta]$ if every message sent from an honest party P at time $T \leq b$ to an honest party Q is received by Q before time $T + \delta$. In this case, for all $T \in [a, b]$, we say that the network is *δ-synchronous at time T*.

2.1.1 Signatures

We make use of an $(n-t)$-out-of-n threshold signature scheme. We refer to a *signature share* and a *signature certificate*: signature shares from $n-t$ on a given message may be combined to form a signature certificate on that message. This can be implemented as (i) a set of signatures, or (ii) an aggregate signature scheme (such as one based on BLS signatures [5] as in [4]), or (iii) a threshold version of an ordinary signature scheme (such as one again based on BLS signatures as in [3]). Implementations (ii) and (iii) will result in much more compact threshold signatures, and (iii) requires a set-up phase to distribute shares of a signing key.

The security property for such a threshold signature scheme is the **Quorum Size Property**: *it is infeasible to produce a signature certificate on a message m, unless $n-t-t'$ honest parties have issued signature shares on m, where $t' \leq t$ is the number of corrupt parties.*

Under our assumption that the number of corrupt parties is strictly less than $n/3$, one can easily establish the following **Quorum Intersection Property**: *it is infeasible to produce signature certificates on two distinct messages m and m', unless at least one honest party issued signature shares on both m and m'.*

2.1.2 Information dispersal

We explicitly make use of well-known techniques for *asynchronous verifiable information dispersal (AVID)* techniques involving erasure codes and Merkle trees (introduced in [6]).

Erasure codes

For integer parameters $k \geq d \geq 1$, a (k,d)-*erasure code* encodes a bit string M as a vector $(f_1,\ldots,f_k)$ of k *fragments* in such a way that any d such fragments may be used to efficiently reconstruct M. Note that for variable-length M, the reconstruction algorithm also takes as input the length β of M. The reconstruction algorithm may fail (for example, a formatting error) – if it fails it returns $\bot$, while if it succeeds it returns a message that when re-encoded will yield k fragments that agree with the original subset of d fragments. We assume that all fragments have the same size, which is determined as a function of k, d, and β.

Using a Reed-Solomon code, which is based on polynomial interpolation, we can realize a (k,d)-erasure code so that if $|M| = \beta$, then each fragment has size $\approx \beta/d$. More precisely, using a Reed-Solomon code over binary finite fields, we can always construct a code such that fragments are of size at most $\max(\lceil \beta/d \rceil, \lceil \log_2(k) \rceil)$ – the term $\lceil \log_2(k) \rceil$ comes from the fact that we need to work with a field of cardinality at least k. In what follows, we will use the more general upper bound of $\beta/d + O(\log(k))$ on fragment size, which serves as an upper bound for the above construction, as well as for other constructions and implementations (which may impose additional restrictions on the length of fragments, such as being a multiple of some specific constant).

In our protocol, the payload of block will be encoded using an $(n, n-2t)$-erasure code. Such an erasure code encodes a payload M as a vector of fragments $(f_1,\ldots,f_n)$, any $n-2t$ of which can be used to reconstruct M. This leads to a data expansion rate of (at most) roughly 3; that is, $\sum_i |f_i| \approx n/(n-2t) \cdot |M| < 3|M|$.

Merkle trees

Recall that a Merkle tree allows one party P to commit to a vector of values $(v_1,\ldots,v_k)$ using a collision-resistant hash function by building a (full) binary tree whose leaves are the hashes of $v_1,\ldots,v_k$, and where each internal node of the tree is the hash of its two children. The root r of the tree is the commitment. Party P may "open" the commitment at a position $i \in [k]$ by revealing v_i along with a "validation path" π_i, which consists of the siblings of all nodes along the path in the tree from the hash of v_i to the root r. We call π_i a *validation path from the root under r to the value v_i at position i*. Such a validation path is checked by recomputing the nodes along the corresponding path in the tree, and verifying that the recomputed root is equal to the given commitment r. The collision resistance of the hash function ensures that P cannot open the commitment to two different values at a given position.

Encoding and decoding

For a given payload M of length β, we will encode M as a vector of fragments $(f_1,\ldots,f_n)$ using an $(n, n-2t)$-erasure code, and then form a Merkle tree with root r whose leaves are the hashes of $f_1,\ldots,f_n$. We define the *tag* $\tau := (\beta, r)$.

For a tag $\tau = (\beta, r)$, we shall call (f_i, π_i) a *certified fragment for τ at position i* if (i) f_i has the correct length of a fragment for a message of length β, and (ii) π_i is a correct validation path from the root under r to the fragment f_i at position i.

The function *Encode* takes as input a payload M. It builds a Merkle tree for M as above with root r (encoding M as a vector of fragments, and then building the Merkle tree whose leaves are the hashes of all of these fragments). It returns $\big(\ \tau,\ \{(f_i,\pi_i)\}_{i\in[n]}\ \big)$, where τ is the tag (β, r), β is the length of M, and each (f_i, π_i) is a certified fragment for τ at position i.

The function *Decode* takes as input $\big(\ \tau,\ \{(f_i, \pi_i)\}_{i \in \mathcal{I}}\ \big)$, where $\tau = (\beta, r)$ is a tag, $\mathcal{I}$ is a subset of $[n]$ of size $n - 2t$, and each (f_i, π_i) is a certified fragment for τ at position i. It first reconstructs a message M' from the fragments $\{f_i\}_{i \in \mathcal{I}}$, using the size parameter β. If $M' = \bot$, it returns $\bot$. Otherwise, it encodes M' as a vector of fragments $(f'_1, \ldots, f'_n)$ and Merkle tree with root r' from $(f'_1, \ldots, f'_n)$. If $r' \neq r$, it returns $\bot$. Otherwise, it returns M'.

Under collision resistance for the hash function used for the Merkle trees, any $n - 2t$ certified fragments for given tag τ will decode to the same payload – moreover, if τ is the output of the encoding function, these fragments will decode to M (and therefore, if the decoding function outputs $\bot$, we can be sure that τ was maliciously constructed). This observation is the basis for the protocols in [11, 18, 26]. Moreover, with this approach, we do not need to use anything like an "erasure code proof system" (as in [2]), which would add significant computational complexity (and in particular, the erasure coding would have to be done using parameters compatible with the proof system, which would likely lead to much less efficient encoding and decoding algorithms).

2.2 Protocol data objects

2.2.1 Blocks

A block B is of the form $\mathsf{Block}(v, v', \tau)$, where (i) $v = 1, 2, \ldots$ is the slot number associated with the block (we say *B is a block for slot v*), (ii) $v' < v$ is the slot number of B's parent block ($v' = 0$ if B's parent is a notional "genesis" block), and (iii) τ is a tag obtained by encoding B's payload M. For simplicity, we call a certified fragment for the tag τ a *certified fragment for B*.

2.2.2 Support, commit, and complaint shares and certificates

A *support share from party P_i on block B* is of the form $\mathsf{SuppShare}(B, \sigma_i, f_i, \pi_i)$, where σ_i is a valid signature share from P_i on $\mathsf{Supp}(B)$, and (f_i, π_i) is a certified fragment for B at position i. A *support certificate on B* is of the form $\mathsf{SuppCert}(B, \sigma)$, where σ is a valid signature certificate on $\mathsf{Supp}(B)$.

A *commit share from party P_i on slot v* is of the form $\mathsf{CommitShare}(v, \sigma_i)$, where σ_i is a valid signature share from P_i on $\mathsf{Commit}(v)$. A *commit certificate on v* is of the form $\mathsf{CommitCert}(v, \sigma)$, where σ is a valid signature certificate on $\mathsf{Commit}(v)$.

A *complaint share from party P_i on slot v* is of the form $\mathsf{ComplaintShare}(v, \sigma_i)$, where σ_i is a valid signature share from P_i on $\mathsf{Complaint}(v)$. A *complaint certificate on v* is of the form $\mathsf{ComplaintCert}(v, \sigma)$, where σ is a valid signature certificate on $\mathsf{Complaint}(v)$.

2.3 Subprotocols

We describe our protocol in terms of a main protocol and a few simple subprotocols. In our presentation, these subprotocols are all running concurrently with each other and with the main protocol: a single party can be thought of as running a local instance of the main protocol and each of the subprotocols on different threads on the same CPU. However, this particular architecture is mainly intended just for ease of presentation. We describe first the data structures and logic of the subprotocols.

2.3.1 Certificate pool

Each party maintains a *certificate pool.* Whenever a party receives a quorum of $n - t$ support, commit, or complaint shares, and it does not already have a corresponding certificate, it will generate a certificate, add it to the pool, and broadcast the certificate to all parties. Similarly, whenever a party receives a support, commit, or complaint certificate, and it does not already have a corresponding certificate, it will add it to the pool, and broadcast the certificate to all parties.

2.3.2 Complete block tree

Each party also maintains a *complete block tree*, which is a tree of blocks rooted at a notional genesis block at slot 0. Under cryptographic assumptions, we will see that there will be at most one block for any given slot in the tree. A block $B = \mathsf{Block}(v, v', \tau)$ is added to the tree if each of the following holds:

- the certificate pool contains a support certificate for B;
- $v' = 0$ or the complete block tree contains a parent block $B' = \mathsf{Block}(v', \cdot, \cdot)$;
- the party has received a quorum of $n - 2t$ support shares for B, from which the party can reconstruct the effective payload M of B as $M \leftarrow \mathit{Decode}(\tau, \{(f_i, \pi_i)\}_{i \in \mathcal{I}})$, where $\{(f_i, \pi_i)\}_{i \in \mathcal{I}}$ is the corresponding collection of certified fragments for τ;
- $M \neq \bot$ and satisfies some correctness predicate that may depend of the path of blocks (and their payloads) from genesis to block B'.

Note that nothing is broadcast when a block is added to the tree.

2.3.3 Block commitment

We say that a block B for slot v is *explicitly committed by party P* if the complete block tree of P contains B and the certificate pool of P contains a commit certificate for slot v. In this case, we say that all of the predecessors of block B in the complete block tree are *implicitly committed by P*. The notional genesis block is always considered to be a committed block. The payloads of committed blocks may be then transmitted in order to the "execution layer" of the protocol stack of a replicated state machine.

2.4 The main protocol

The logic of the main protocol for a party P_j is described in Fig. 1. In the description, $\mathsf{leader}(v)$ denotes the leader for slot v – as discussed above, leaders may be rotated in each slot, either in a round-robin fashion or using some pseudo-random sequence. The details for generating and validating block proposals are described below. In the main protocol, a party makes its decisions based on the objects in its certificate pool and its complete block tree (which are maintained as described in Section 2.3) and the objects it has received from other parties over authenticated channels. The core of the protocol is expressed in terms of a "wait until either" statement which triggers one of several clauses based different preconditions. Although not strictly necessary, for concreteness, we assume that if more than one clause's precondition is satisfied, then the syntactically first such clause is triggered.

The basic idea is this. The leader for slot v will send each party a block proposal for a block B. Upon receiving such a block proposal, each party validates the proposal and then sends a support share for B to all parties. Each party will move onto the next slot when it adds B to its complete block tree; however, if too much time elapses before that happens, it will broadcast a complaint share, and move onto the next slot when it obtains a corresponding complaint certificate. Note that when a party moves onto the next slot by virtue of adding B to its complete block tree, it will also issue a commit share for v, but only if it has not already issued a complaint share for slot v – this rule is essential for safety.

DispersedSimplex: *main loop for party P_j*

```
$v_{\text{last}} \leftarrow 0$
for $v = 1, 2, \ldots$
    $T_{\text{start}} \leftarrow \mathsf{clock}()$,  $done \leftarrow proposed \leftarrow supported \leftarrow complained \leftarrow \mathsf{false}$
    while not $done$ do
        wait until either:
            the certificate pool contains a complaint certificate for slot $v \Rightarrow$
                $done \leftarrow \mathsf{true}$
            the complete block tree contains a block for slot $v \Rightarrow$
                if not $complained$ then broadcast a commit share for $v$
                $done \leftarrow \mathsf{true}$, $v_{\text{last}} \leftarrow v$
            not $complained$ and $\mathsf{clock}() > T_{\text{start}} + \Delta_{\text{timeout}} \Rightarrow$
                $complained \leftarrow \mathsf{true}$,  broadcast a complaint share for slot $v$
            $\mathsf{leader}(v) = P_j$ and not $proposed \Rightarrow$
                $proposed \leftarrow \mathsf{true}$
(*)             generate block proposal material $B$, $(f_1, \pi_1), \ldots, (f_n, \pi_n)$
                for $i \in [n]$: send $\mathsf{BlockProp}(B, f_i, \pi_i)$ to $P_i$
(**)        not $supported$ and received from $\mathsf{leader}(v)$ a valid block proposal
            $\mathsf{BlockProp}(B, f_j, \pi_j) \Rightarrow$
                $supported \leftarrow \mathsf{true}$
                generate a signature share $\sigma_j$ on $\mathsf{Supp}(B)$
                broadcast the support share $\mathsf{SuppShare}(B, \sigma_j, f_j, \pi_j)$
```

Figure 1 Logic for main loop of DispersedSimplex protocol for party P_j

2.4.1 Generating block proposals

The logic for generating block proposal material B, $(f_1, \pi_1), \ldots, (f_n, \pi_n)$ in slot v at line $(*)$ is as follows: (i) build a payload M that validly extends the path in the complete block tree ending at the block for slot v_{last}; (ii) compute $(\tau, \{(f_i, \pi_i)\}_{i \in [n]}) \leftarrow Encode(M)$; (iii) set $B \coloneqq \mathsf{Block}(v, v_{\text{last}}, \tau)$.

2.4.2 Validating block proposals

To check if $\mathsf{BlockProp}(B, f_j, \pi_j)$ is a valid block proposal from the leader in slot v at line $(**)$, party P_j checks that each of the following conditions holds: (i) B is of the form $\mathsf{Block}(v, v', \tau)$, where $v' < v$ and the complete block tree contains a block for slot v'; (ii) the certificate pool contains complaint certificates for slots $v' + 1, \ldots, v - 1$; (iii) (f_j, π_j) is a certified fragment for τ at position j.

Note that even if some of the conditions do not hold at a given point in time, they may hold at a later point in time. When party P_j sees a block proposal in slot v, it can check the stated conditions – if these conditions fail due to the lack of either a parent block in the complete block tree or a complaint certificate, these conditions will need to be rechecked whenever a new block is added to the complete block tree or a new complaint certificate is added to the certificate pool. We will discuss below (in Appendix B.1) how to efficiently implement the test that the certificate pool contains the necessary complaint certificates using a data structure whose size is proportional to the gap between current slot and the last committed slot so that the amortized cost of these tests is $O(1)$ per slot.

3 Analysis

By abuse of terminology, we state security properties unconditionally – they implicitly assume the security of the threshold signature scheme and the collision resistance of the hash functions used to build Merkle trees, and should be understood to hold with all but negligible probability for all efficient adversaries.

3.1 Initial observations

We state some basic properties:

Uniqueness and Validity Property: Suppose that a block B for some slot v is added to the complete block tree of some party. Then no other block for slot v can be added to the complete block tree of that party or any other party. Moreover, if the leader for slot v is honest, B must have been proposed by that leader.

The first part follows from the Quorum Intersection Property, based on the fact an honest party issues a support share for at most one block per slot. The second part follows from the Quorum Size Property.

Completeness Property: If an object X appears in the certificate pool (so X is a support, commit, or complaint certificate) or in the complete block tree (so X is a block), then X (or its equivalent) will eventually appear in the corresponding pool/tree of every other party.[1] Moreover, if X appears in a party's pool/tree at a time T at which the network is δ-synchronous, it will appear in every party's pool/tree before time $T + \delta$.

For the support, commit, and complaint certificates, this is clear. For the blocks in the complete block tree, we are relying on the Quorum Size Property: when a support certificate for a block B is added to the support pool, at least $n - 2t$ honest parties must have already broadcast support shares for B, which contain B as well as fragments sufficient to reconstruct B's payload.

Incompatibility of Complaint and Commit Property: It is impossible to produce both a complaint and commit certificate for the same slot v.

This follows from the Quorum Intersection Property, based on the fact that in each slot, an honest party will never issue both a complaint share and a commit share.

3.2 Safety

Safety follows immediately from the following lemma. See Appendix A.1 for a proof.

▶ **Lemma 1** (Safety). *Suppose a party P explicitly commits a block B for slot v, and a block C for slot $w \geq v$ is in the complete block tree of some party Q. Then B is an ancestor of C in Q's complete block tree.*

3.3 Liveness

Liveness follows immediately from the following lemmas. The first lemma analyzes the optimistic case where the network is synchronous and the leader of a given slot is honest, showing that the leader's block will be committed. See Appendix A.2 for a proof.

[1] Note that the "or equivalent" qualification is necessary to account for signature certificates, if these are not necessarily unique.

▶ **Lemma 2** (Liveness I). *Consider a particular slot $v \geq 1$ and suppose the leader for slot v is an honest party Q. Suppose that the first honest party P to enter the loop iteration for slot v does so at time T. Further suppose that the network is δ-synchronous over the interval $[T, T + 3\delta]$ for some δ with $\Delta_{\text{timeout}} \geq 3\delta$. Then each honest party will finish the loop iteration before time $T + 3\delta$ by adding Q's proposed block B to its complete block tree. and will eventually commit B. Moreover, each honest party will eventually commit B, and this will happen before time $T + 4\delta$ if the network remains δ-synchronous over the interval $[T, T + 4\delta]$.*

The second lemma analyzes the pessimistic case, when the network is asynchronous or the leader of a given round is corrupt. It says that eventually, all honest parties will move on to the next round. See Appendix A.3 for a proof.

▶ **Lemma 3** (Liveness II). *Suppose that the network is δ-synchronous over an interval $[T, T + \Delta_{\text{timeout}} + 2\delta]$, for an arbitrary value of δ, and that at time T, some honest party is in the loop iteration for slot v and all other honest parties are in a loop iteration for v or a previous slot. Then before time $T + \Delta_{\text{timeout}} + 2\delta$, all honest parties finish the loop iteration for slot v.*

We note that in periods of asynchrony, for any slot v in which the leader Q is honest, if any block is committed in slot v, it must have been the block proposed by Q. This follows from the (second part of the) Uniqueness and Validity Property.

We also remark that by the Incompatibility of Complaint and Commit Property, for a valid block B, the slot number of B's parent block cannot be less than that of the last committed block. This property is enables a practical implementation to keep the storage bounded using standard techniques of checkpointing and garbage collection (and perhaps with standard techniques for dynamically increasing timeouts until commitments are seen).

3.4 Complexity estimates

3.4.1 Communication complexity

We measure the communication complexity per slot. This is the sum over all honest parties P and all parties Q of the bit-length of all slot-v-specific messages sent from P to Q. The communication complexity per slot of DispersedSimplex is easily seen to be bounded by $3n\beta + O(n^2(\kappa + \lambda \log n))$, where (i) β is a bound on the size of a block, (ii) κ is a bound on the size of a threshold signature share or certificate, and (iii) λ is a bound on the size of the hash function outputs used for Merkle trees. Indeed, the cost breaks down as follows: (i) $3n\beta + O(n^2 \log n)$ for disseminating payload fragments, (ii) $O(n^2 \log n \cdot \lambda)$ for disseminating Merkle paths, and (iii) $O(n^2\kappa)$ for disseminating signature shares and certificates. If blocks are large, in particular, if $\beta \gg n(\kappa + \lambda \log n)$, the communication complexity will be dominated by the cost of disseminating the payload fragments.

Moreover, the communication load is *balanced*, meaning that each party, *including the leader for a slot*, transmits roughly the same about of data over the network. In fact, as we described the protocol, for large β, each non-leader transmits about 3β bits in total, while the leader transmits about 6β bits in total. In Section 3.5, we discuss a simple variation in which the leader also transmits only about 3β bits. In the full version [22], we discuss a variation in which each party transmits only 1.5β–2β bits.

3.4.2 Latency

We may also measure various notions of latency. We define:

- *optimistic proposal-commit latency:* assuming the leader is honest, and that the network is appropriately synchronous, the time it takes for the leader's proposal to be committed by all honest parties (same as the notion of "proposal confirmation time" in [9]);
- *optimistic consecutive-proposal latency:* assuming two consecutive leaders are honest, and that the network is appropriately synchronous, the amount of time that elapses between when they make their respective proposals (similar to the notion of "optimistic block time" in [9]).

If a given transaction is submitted to the system (i.e., to all parties), the sum of these two latencies upper bounds the total time it takes for a transaction to be included in a proposal and then committed. The optimistic consecutive-proposal latency also upper bounds what we might call the *optimistic reciprocal block throughput*, the reciprocal of the rate at which blocks are proposed (and committed) in a steady state where all leaders are honest and the network is appropriately synchronous.

For DispersedSimplex, just as for Simplex, we readily see that if the network is δ-synchronous with $\Delta_{\text{timeout}} \geq 3\delta$, then the optimistic proposal-commit latency is 3δ and the optimistic consecutive-proposal latency is 2δ. This proposal-commit latency is optimal (it matches lower bound in [1] for psync-BB).

It is also useful to look at the latency between proposals made between non-consecutive honest leaders. That is, if leaders in slots v and $v + k + 1$ are honest, but the k leaders in the intervening slots are crashed or corrupt, how much time may elapse between the time the leader in slot v makes its proposal and the time the leader in slot $v + k + 1$ makes its proposal. Let us call this the *optimistic k-gap proposal latency.* For DispersedSimplex, just as for Simplex, this is $2\delta + k \cdot (\Delta_{\text{timeout}} + \delta)$. If leaders are chosen at random, then the probability that there is a gap of size k between slots with honest leaders decreases exponentially with k. We note that DispersedSimplex protocol is *optimistically responsive*, meaning that it runs as fast as the network will allow so long as leaders are honest.

3.5 Other costs and concrete estimates

The above analysis abstracts away a number of practically important details. Indeed, our latency estimates in Section 3.4.2 only took into account propagation delays caused by network latency, but did not take into account transmission delay (caused by limited network bandwidth) and computation delay (caused by limited compute bandwidth).

In this section, we discuss other costs and make some concrete estimates for performance under specific assumptions. We are generally interested in values of n up to around 100, where each of the n parties is running commodity hardware and connected to a WAN with typical network bandwidth and latency.

We first consider the computational cost of erasure coding. This should not have a significant impact on the overall system performance, assuming one uses a reasonably good implementation of erasure coding algorithms. One such implementation is the `reed-solomon-simd` library at `https://github.com/AndersTrier/reed-solomon-simd`, which is based on [15, 14]. We benchmarked this implementation with parameters corresponding to $t = 32$ and $n = 3t+1 = 97$ and payload sizes of 100KB and 1MB on a Macbook Pro with an Apple M1 Max CPU. The encoder runs at a rate of nearly 2GB/s for both payload sizes. The decoder runs at a rate of about 250MB/s for the 100KB payload and about 500MB/s per second for the 1MB payload. Generally, the encoder speed is independent of the payload size and the decoder speed increases with the payload size (because fixed costs get amortized). At these speeds, it is very unlikely that the erasure coding will be a bottleneck.

We next consider the computational cost of signature generation, verification, and aggregation. Let us assume we use aggregate BLS signatures with the standard proof-of-possession mitigation against rogue-key attacks, so that public keys and signatures are very cheaply aggregated by simply adding them together. On the same hardware above, we benchmarked the `blst` library at `https://github.com/supranational/blst`. The cost of signing or verifying one BLS signature is well under 1ms, and the cost of adding public keys and signatures in the aggregation process can be effectively ignored (at least for quorums of size up to a few hundred). To aggregate many unverified BLS signatures, a party P can very cheaply aggregate the unverified signatures and then verify the result. If the aggregate verification fails, P will have to perform a much more expensive search to find out which of the individual signatures were bad. However, once the bad signatures are found, since the parties that contributed those signatures must be corrupt, P can simply ignore all signatures (and indeed all messages) sent from these parties going forward. This works because we are assuming the signatures are sent over authenticated channels (although P cannot publicly prove their corrupt behavior, unless the BLS signatures are themselves authenticated using some cheaper digital signature, such as EdDsa). Thus, over the long run, the cost of verifying and aggregating a set of individual signatures is essentially just the cost of one BLS signature verification. Similarly, when a party P receives an aggregate signature from another party, if the verification of that aggregate signature fails, P can simply ignore that party going forward.

The other main computational cost to consider is that of hashing. On the same hardware mentioned above, the `openssl` implementation of SHA256 runs at a speed of 2GB/s.

With these benchmarks, and additional assumptions on network bandwidth and latency, we can estimate the performance (latency and throughput) of the protocol (in the optimistic setting). We shall assume network bandwidth of 1Gb/s (i.e., 125MB/s) and that the protocol is running over a WAN, so that there is essentially no contention for network bandwidth among the parties. Specifically, our assumption is that all parties can simultaneously transmit to the network at a rate of 1Gb/s. We shall assume a network latency of 100ms (so it takes 100ms for a packet to travel from P to Q once P has transmitted the packet, which is generally consistent with round-trip times reported in `https://www.cloudping.co/grid/p_90/timeframe/1D`).

The protocol's performance will depend on: (i) *transmission delay*, the delay per slot induced by network bandwidth, (ii) *propagation delay*, the delay per slot induced by the network latency, and (iii) *computation delay*, the delay induced by computation. The optimistic consecutive-proposal latency is just the sum of these delays and throughput is the block size β divided by the sum of these delays. Here, we will assume that β is the number of *bytes* in a block. Of course, β also impacts transmission and computation delay.

We will make one small change to the protocol that will streamline its execution. Namely, instead of using an $(n, n-2t)$-erasure code, we will use an $(n-1, n-2t-1)$-erasure code, and adopt the convention that the leader does not hold a fragment. We note that with this change, the encoding of a block is still at most 3β bytes, and that the above benchmarks for $n = 97$ are still valid. With this change, the way the block data flows through the network in a given slot is as follows:

- the leader encodes a block of size β as a codeword of size $\approx 3\beta$, and transmits to each of the $n-1$ other parties its fragment, which has size $\approx 3\beta/n$, so that the leader transmits a total of $\approx 3\beta$ bytes across the network.

- each party other than the leader broadcasts its fragment of size $\approx 3\beta/n$ to the $n-2$ other parties (besides itself and the leader), so each such party transmits a total of $\approx 3\beta$ bytes across the network.

Assuming fragments are sufficiently large, each fragment can be broken up into many packets, and a simple "packet-switching pipeline" strategy can be used to minimize the transmission delay. Specifically, the leader begins by sending to each other party P the first packet of P's fragment, then it sends to each other party P the second packet of P's fragment, and so on; at the same time, when a party P receives one packet of its own fragment from the leader, it immediately broadcasts that fragment to all other parties. One sees that with this simple "packet-switching pipeline" strategy, the transmission delay per slot is roughly 3β bytes divided by the network bandwidth available to each party (without pipelining, it would be twice as much). With a network bandwidth of 1Gb/s, this translates into a transmission delay per slot of about 25ms for every 1MB of (original, unencoded) block data.

Next, consider propagation delay. This is twice the network latency, so $2 \cdot 100\text{ms} = 200\text{ms}$ under our assumptions. To make things more concrete, let us choose a block size that roughly balances transmission and propagation delay, so a block size of 8MB. With a block size this large, and for $n \approx 100$, the size of each fragment is $\approx$ 240KB, large enough to make the simple "packet-switching pipeline" strategy feasible (with packets of size $\approx 1KB$, a party can transmit one packet to each other party in time under 1ms).

Third, consider computation delay. There are several components to this:

- *erasure coding:* the leader encodes β bytes of data, and then each receiving party decodes and encodes the same amount of data; with our given estimates (for $n = 97$), this takes $2 \cdot 4\text{ms} + 16\text{ms} = 24\text{ms}$. Using multiple cores, this could likely be reduced significantly.
- *hashing:* the leader hashes 3β bytes of data, and then each receiving party hashes the same amount of data; with our given estimates, this takes $2 \cdot 12\text{ms} = 24\text{ms}$. However, the hashing done by the leader can overlap entirely with the transmission delay (the hashing can be done concurrently with the transmission of the fragments). For the receiving parties, in a typical execution, of the 3β bytes of data they need to hash, at least 2β bytes of hashing can overlap with the transmission delay (assuming the hashing is done as packets are received). If they receive support shares from all other parties, no more hashing needs to be done. In the worst case, they need to hash β bytes (after the re-encoding step), and with our given estimates, this takes 4ms. Using multiple cores, this could likely be reduced even more.
- *signing and aggregating:* each party generates a support share and then forms a support certificate. With our given estimates, this takes a total of 2ms. However, the 1ms of time spent forming a support certificate easily overlap the above 4ms of hashing time (assuming multiple cores). We do not count here the cost of processing commit shares and certificates, as these can be performed on a separate core.

This all adds up to a computation delay of $24\text{ms} + 4\text{ms} + 1\text{ms} = 29\text{ms}$, and we will round this up to 40ms to be conservative (although by exploiting multiple cores, it could be much less).

With these parameters, we estimate the total delay per slot as: 200ms for transmission, 200ms for propagation, 40ms for computation. This translates to a throughput of 8MB every 440ms, so about 18MB per second. The optimistic consecutive-proposal latency is 440ms and the optimistic proposal-commit latency is that plus about 100ms, so about 540ms.

To get a better understanding of this setting, consider the following example timeline. Suppose that at time T a leader starts transmitting the packets of a block. By time (roughly) $T + 100\text{ms}$ the other parties start echoing these packets. By time (again, roughly) $T + 200\text{ms}$ the leader finishes transmitting packets and transmits the remaining elements of its block proposal. By time $T + 300$ms all of these packets and remaining elements have been echoed by the other parties; moreover, by this same time, the other parties have validated the block proposal and have broadcast a signature share on a corresponding support message. By time $T + 400$ms, the other parties have received all the fragments and other data they need, and then perform 40ms of computation to finish the slot with a block in the complete block tree by time $T + 440$ms.

Note that all of the above estimates are essentially independent of n. Indeed, the component of propagation and computation delay that depends on n will be a very small fraction of the total for block sizes of at least 1MB and for n up to several hundred.

Appendix B briefly presents some minor implementation details and simple variations of DispersedSimplex.

4 Stable leaders

In many settings, it makes sense to keep a leader that is doing a good job in place for an extended number of slots. There are a number of advantages to this. For example, whenever such a crashed party is selected as a leader, the protocol has to wait sufficiently long to “time out” and move to the next slot, effectively wasting the equivalent of a few slots. In contrast, if a leader by default stays in place for, say, 1000 slots, when we come to a crashed leader, we will still waste the equivalent of a few slots, but this will be a much smaller percentage of all slots. Another advantage is that if transactions are being submitted to the system by external clients, then (just as in classical PBFT) these transactions can typically just be sent to a stable leader. Yet another advantage, as we will discuss below, is that a stable leader can drive the protocol even faster, achieving both higher throughput and lower latency.

The Simplex protocol has such a very natural internal logic to it that the logic for maintaining stable leaders suggests itself almost immediately. Let us say that by default a leader will stay in place for a certain number of consecutive slots, which we call an *epoch*. For example, one epoch might be 1000 consecutive slots.

- So that we can move to the next epoch as soon as we detect a faulty leader, we shall adopt the convention that a complaint certificate for a slot v effectively covers the rest of the epoch containing v.
- In order to maintain safety, this means that any party that issues a complaint share for a slot v must abstain from issuing a commit certificate in slot v *and all remaining slots of the interval containing v*.
- This means that once one honest party issues a complaint share for a slot v, it may not be possible to commit a block in slot v *or in any of the remaining slots of the interval containing v*, even though blocks may continue to be supported and added to the complete block tree.
- Therefore, in order to maintain liveness, we introduce logic that prevents parties from moving too far ahead of the slot of the last committed block in an epoch.

The details of our protocol, which we call **StableDispersedSimplex**, are in Fig. 2. Note that for any slot number v, $\mathsf{begin}(v)$ denotes the first slot number of the epoch containing v, while $\mathsf{end}(v)$ denotes the last slot number in an epoch. The value k is a constant parameter, which can be set to 1 or any other small positive integer. The logic to go to the next slot on seeing an approved block ensures that the approved blocks do not get more than k slots ahead of the committed blocks (and if the network is well behaved and the leader is honest, it should never get more than 1 slot ahead).

```
StableDispersedSimplex: main loop for party $P_j$

$v_{\text{last}} \leftarrow 0$, $v \leftarrow 1$
repeat forever
    $T_{\text{start}} \leftarrow \mathsf{clock}()$,  $\mathit{done} \leftarrow \mathit{proposed} \leftarrow \mathit{supported} \leftarrow \mathit{complained} \leftarrow \mathsf{false}$
    if $v = \mathsf{begin}(v)$ then $\mathit{complainedInEpoch} \leftarrow \mathsf{false}$   // new epoch
    while not $\mathit{done}$ do
        wait until either:
            the certificate pool contains a complaint certificate for any slot in $[\mathsf{begin}(v) \,..\, v] \Rightarrow$
                $\mathit{done} \leftarrow \mathsf{true}$, $v \leftarrow \mathsf{end}(v) + 1$ // go to next epoch
            the complete block tree contains a block for slot $v$ and
              ( $v = \mathsf{end}(v)$ or there is a committed block for all slots in $[\mathsf{begin}(v) \,..\, v - k]$ ) $\Rightarrow$
                if not $\mathit{complainedInEpoch}$ then broadcast a commit share for $v$
                $\mathit{done} \leftarrow \mathsf{true}$, $v_{\text{last}} \leftarrow v$, $v \leftarrow v + 1$   // go to next slot
            not $\mathit{complained}$ and ( $\mathit{complainedInEpoch}$ or $\mathsf{clock}() > T_{\text{start}} + \Delta_{\text{timeout}}$ ) $\Rightarrow$
                $\mathit{complained} \leftarrow \mathit{complainedInEpoch} \leftarrow \mathsf{true}$
                broadcast a complaint share for slot $v$
            // The rest is the same as in Fig. 1
```

Figure 2 Logic for main loop of StableDispersedSimplex protocol for party P_j

The protocol makes use of the identical subprotocols for maintaining the certificate pool and complete block tree. The logic for generating block proposals is identical to that in the basic protocol.

The logic for validating block proposals is the same as in the basic protocol, except as follows. First, if $v > \mathsf{begin}(v)$, we require that $v = v' + 1$, which enshrines the fact that an honest leader should propose blocks with consecutive slot numbers. Second, instead of checking that the certificate pool contains complaint certificates for slots $v' + 1, \ldots, v - 1$, we check that it contains complaint certificates that effectively cover this interval – that is, for each $w \in [v' + 1 \,..\, v - 1]$, there exists a complaint certificate for a slot u such that $w \in [u \,..\, \mathsf{end}(u)]$. It is an easy exercise to generalize the data structures and algorithms in Appendix B.1 to work in this setting. One sees that this protocol is identical to the basic protocol if all epochs are of size 1.

4.1 Analysis

We sketch here the main ideas of the safety and liveness analysis for this protocol.

The basic properties in Section 3.1 hold here as well, except that the *Incompatibility of Complaint and Commit Property* generalizes here as follows: if a complaint certificate for a slot v has been produced, then it is impossible to produce a commit certificate for any slot in $[v \,..\, \mathsf{end}(v)]$. This follows from the Quorum Intersection Property and the fact that if an honest party issues a complaint share in slot v, it will not issue a complaint share in v or any subsequent round in the same epoch as v.

Lemma 1 holds for this protocol as stated. The proof of Lemma 1 go through with essentially no change, other than to note the fact that we use complain certificates that cover the interval $[v' + 1 \,..\, v - 1]$. Lemma 2 may be adjusted as follows (proof in Appendix A.4):

▶ **Lemma 4** (Liveness I – stable leader version). *Consider a particular slot $v \geq 1$ and suppose the leader for slot v is an honest party Q. Suppose that the first honest party P to enter the loop iteration for slot v does so at time T. Further suppose that the network is δ-synchronous over the interval $[T, T + 3\delta]$ for some δ with $\Delta_{\text{timeout}} \geq 3\delta$. Then before time $T + 3\delta$, each honest party will reach a loop iteration $\geq v$. In addition, if each honest party issues a commit* *share in rounds* $\mathsf{begin}(v), \ldots, v-1$*, then each honest party will finish loop iteration* v *before time* $T + 3\delta$*, by adding* Q*'s proposed block* B *to its complete block tree and issuing a commit share for round* v*.*

The lemma is stated as it is so that by repeated application of the lemma, it follows that so long as the network remains appropriately synchronous, an honest leader will continue to get all of its proposals committed.

Lemma 3 holds for this protocol essentially as stated – the conclusion would be better worded as "all honest parties have entered the loop iteration for some slot $w > v$". The proof only needs to be changed to reflect the fact that before time $T + \delta$, every honest party either enters the loop iteration for slot v or moves to the next epoch because of a complaint certificate for some round in $[\mathsf{begin}(v) \mathinner{..} v-1]$. In the latter case, before time $T + 2\delta$, all honest parties will have moved to the next epoch.

4.2 Improved performance through stability

As mentioned above, performance can be improved by having stable leaders. To see how, let us return to the concrete example in Section 3.5, with the parameters used there: $n \approx 100$ parties connected over a WAN, 1Gb/s bandwidth, 100ms latency, and an 8MB block size.

In the example timeline we gave there, if the leader starts transmitting the packets of a block at time T, then by time (roughly) $T + 200$ms the leader stops transmitting, but the other parties will not finish the slot until time (again, roughly) $T + 440$ms. With a constantly rotating leader, the leader for the next slot will wait until this time before it begins transmitting the packets of its block. However, a stable leader can start transmitting these packets already at time $T + 200$ms. Indeed, between time T and $T + 200$ms, it could have gathered the transactions for its next block (and even performed the erasure encoding of that block), so that it can start transmitting the these packets right away at time $T + 200$ms.

Thus, throughout an epoch where the leader is honest and the network is synchronous, we basically get another level of pipelining, with the leader starting a new slot every 200ms. Note that in these circumstances, all parties will essentially fully utilize all available network bandwidth. (Achieving all this assumes multi-threading on a few cores.) This translates to a throughput of 8MB every 200ms, so about 40MB per second. The optimistic consecutive-proposal latency is 200ms. The optimistic proposal-commit latency remains the same as in the rotating leaders version, so about 540ms.

Finally, we note that while the stable leader may nearly saturate its upload bandwidth, it is not consuming very much download bandwidth, which leaves plenty of bandwidth available for downloading transactions that are submitted directly to the stable leader by external clients.

See Appendix B.3 for simple variations on StableDispersedSimplex. In the full version [22], we (i) give a more extended version of the above example timeline, (ii) show how to double the throughput to 80MB per second using the improved data dissemination techniques in [17] without impacting latency, and (iii) discuss performance, quality, and censorship attacks on StableDispersedSimplex, and ways of mitigating against them.

References

1 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization, 2021. arXiv:2102.07240, http://arxiv.org/abs/2102.07240.

2 Nicolas Alhaddad, Sisi Duan, Mayank Varia, and Haibin Zhang. Succinct erasure coding proof systems. Cryptology ePrint Archive, Paper 2021/1500, 2021. URL: https://eprint.iacr.org/2021/1500.

3 Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Yvo Desmedt, editor, *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, volume 2567 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2003. doi:10.1007/3-540-36288-6_3.

4 Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. Cryptology ePrint Archive, Paper 2018/483, 2018. URL: https://eprint.iacr.org/2018/483.

5 Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer, 2001. doi:10.1007/3-540-45682-1_30.

6 Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In Pierre Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, volume 3724 of *Lecture Notes in Computer Science*, pages 503–504. Springer, 2005. doi:10.1007/11561927_42.

7 Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet computer consensus. Cryptology ePrint Archive, Report 2021/632, 2021. URL: https://ia.cr/2021/632.

8 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999. URL: https://dl.acm.org/citation.cfm?id=296824.

9 Benjamin Y. Chan and Rafael Pass. Simplex consensus: A simple and fast consensus protocol. In Guy N. Rothblum and Hoeteck Wee, editors, *Theory of Cryptography - 21st International Conference, TCC 2023*, volume 14372 of *Lecture Notes in Computer Science*, pages 452–479. Springer, 2023. Also at https://eprint.iacr.org/2023/463. doi:10.1007/978-3-031-48624-1_17.

10 George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 34–50. ACM, 2022. Also at arXiv:2105.11827, http://arxiv.org/abs/2105.11827. doi:10.1145/3492321.3519594.

11 S. Dolev and Z. Wang. SodsBC: Stream of distributed secrets for quantum-safe blockchain. In *2020 IEEE International Conference on Blockchain (Blockchain)*, pages 247–256, Los Alamitos, CA, USA, 2020. IEEE Computer Society.

12 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988. doi:10.1145/42282.42283.

13 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 165–175. ACM, 2021. Also at arXiv:2102.08325, http://arxiv.org/abs/2102.08325. doi:10.1145/3465084.3467905.

14 Sian-Jheng Lin, Tareq Y. Al-Naffouri, Yunghsiang S. Han, and Wei-Ho Chung. Novel polynomial basis with fast Fourier transform and its application to Reed-Solomon erasure codes. *IEEE Trans. Inf. Theory*, 62(11):6284–6299, 2016. doi:10.1109/TIT.2016.2608892.

15 Sian-Jheng Lin and Wei-Ho Chung. An efficient (n, k) information dispersal algorithm for high code rate system over Fermat fields. *IEEE Commun. Lett.*, 16(12):2036–2039, 2012. doi:10.1109/LCOMM.2012.112012.121322.

16 Thomas Locher. Byzantine reliable broadcast with low communication and time complexity, 2024. arXiv:2404.08070, http://arxiv.org/abs/2404.08070.

17 Thomas Locher and Victor Shoup. MiniCast: Minimizing the communication complexity of reliable broadcast. Cryptology ePrint Archive, Paper 2024/571, 2024. URL: https://eprint.iacr.org/2024/571.

18 Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-MVBA: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In Yuval Emek and Christian Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 129–138. ACM, 2020. doi:10.1145/3382734.3405707.

19 Dahlia Malkhi and Kartik Nayak. Extended abstract: HotStuff-2: Optimal two-phase responsive BFT. Cryptology ePrint Archive, Paper 2023/397, 2023. URL: https://eprint.iacr.org/2023/397.

20 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 31–42. ACM, 2016. Also at https://eprint.iacr.org/2016/199. doi:10.1145/2976749.2978399.

21 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990. doi:10.1145/98163.98167.

22 Victor Shoup. Sing a song of simplex. Cryptology ePrint Archive, Paper 2023/1916, 2023. URL: https://eprint.iacr.org/2023/1916.

23 Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness, 2023. arXiv:2306.03058, http://arxiv.org/abs/2306.03058.

24 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2705–2718. ACM, 2022. Also at arXiv:2201.05677, http://arxiv.org/abs/2201.05677. doi:10.1145/3548606.3559361.

25 Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. Mir-BFT: High-throughput robust bft for decentralized networks, 2019. arXiv:1906.05552, http://arxiv.org/abs/1906.05552.

26 Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. DispersedLedger: High-throughput byzantine consensus on variable bandwidth networks, 2021. arXiv:2110.04371, http://arxiv.org/abs/2110.04371.

27 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus in the lens of blockchain, 2018. arXiv:1803.05069, http://arxiv.org/abs/1803.05069.

A Some proofs

A.1 Proof of Lemma 1

By the Incompatibility of Complaint and Commit Property, no complaint certificate for slot v can be produced. Let C' be the parent of C and suppose w' is the slot number of C'. Since C' is in Q's complete block tree, a support certificate for C' must have been produced, which means at least one honest party must have issued a support share for C', which means $v \leq w' < w$. The inequality $v \leq w'$ follows from the fact that there is no complaint certificate for slot v, and an honest party will issue a support share for C only if it has complaint certificates for slots $w' + 1, \ldots, w - 1$.

If $v = w'$, we are done by the (first part of the) Uniqueness and Validity Property, and if $v < w'$, we can repeat the argument inductively with C' in place of C.

A.2 Proof of Lemma 2

By the Completeness Property, before time $T + \delta$, each honest party will enter the loop iteration for slot v by time $T + \delta$, having either a complaint certificate for slot $v - 1$ or a block for slot $v - 1$ in its complete block tree. So before time $T + \delta$, the leader Q will propose a block B that extends a block B' with slot number $v' < v$. By the logic of the protocol, we know that Q must have complaint certificates for slots $v' + 1, \ldots, v - 1$ at the time it makes its proposal. Again by the Completeness Property, before time $T + 2\delta$, each honest party will have B' in its complete block tree and all of these complaint certificates in its certificate pool, and moreover, will receive Q's proposal before this time, and hence will broadcast a support share for Q's proposal by this time. Therefore, before time $T + 3\delta$, each honest party will have added B to its complete block tree. By the assumption that $\Delta_{\text{timeout}} \geq 3\delta$, when each honest party adds B to its complete block tree, the complaint condition will not have been met, and therefore, each honest party will issue a commit share for v at this time. If the network remains δ-synchronous, the commit shares will be received by all honest parties before time $T + 4\delta$.

A.3 Proof of Lemma 3

By the Completeness Property, every honest party will enter the loop iteration for slot v before time $T + \delta$. By time $T + \delta + \Delta_{\text{timeout}}$, every honest party will have either added a block for slot v to its complete block tree or broadcast a complaint share for slot v. In either case, less than δ time units later all honest parties will have finished the loop iteration for slot v.

A.4 Proof of Lemma 4

The proof goes through with essentially no change in the case where v is the first slot in an epoch. For later slots in the epoch, we need to add the extra assumption that each honest party issued a commit share for all previous slots in the epoch – and so did not issue a complaint share in those slots. This guarantees that before time $T + \delta$ all honest parties will enter the loop iteration for slot v, and that before time $T + 2\delta$, not only will all honest parties issue support shares for Q's proposal but will also commit the block for slot $v - 1$. Therefore, before time $T + 3\delta$, each party will finish the loop iteration for slot v as stated.

B Some implementation details and minor variations

B.1 Implementing the block proposal validation logic

To validate a proposal for a block B in slot v whose parent is a block B' in slot v', a party needs to check if its complaint pool contains complaint certificates for slots $v' + 1, \ldots, v - 1$. Here is a simple, practical way to do this.

Suppose that when a party enters the loop iteration for slot v, the highest slot number for which it has committed is v_{com}. We know by the Incompatibility of Complaint and Commit Property, there can never be a complaint certificate for slot v_{com}. So the party can maintain two data structures.

- A doubly linked list of those slots in the range $\{v_{\text{com}}, \ldots, v-1\}$ for which it *does not* have a complaint certificate, in order from lowest to highest.
- A lookup table from $\{v_{\text{com}}, \ldots, v-1\}$ to nodes in this doubly linked list – this table could just be a dynamic, circular array.

Then, the party can perform the following operations:

- Whenever a new complaint certificate appears for a slot in the range $\{v_{\text{com}}, \ldots, v-1\}$, it accesses the corresponding node via the lookup table and removes it from the linked list.
- When the value of v_{com} or v is increased, it updates both the lookup table and linked list in the obvious way.

For each slot, a constant amount of work is performed to maintain this data structure. Moreover, at any point in time, a party can find in constant time the highest slot number $v^* < v$ for which it has complaint certificates for slots $v^*+1, \ldots, v-1$.

B.2 Simple variations

We mention here a few simple variations of DispersedSimplex.

- *Choice of parent block.* In the protocol, the leader in slot v proposes a new block whose parent is B_{prev}. In fact, the leader is free to choose as the parent block any block B' for a slot v' such that $v' < v$ and the leader's complaint pool contains complaint certificates for each slot $v'+1, \ldots, v-1$.
- *Moving on from bad blocks.* In the protocol, in managing the complete block tree, when a party reconstructs the payload and finds that it is bad (either $\perp$ or otherwise invalid), it effectively just ignores the block and the slot will eventually time out. In a variation, parties could simply issue a complaint share right away, without waiting to time out.
- *Withholding support after complaining.* As we described the protocol, a party may issue a support share in a slot even if it has already issued a complaint share in that slot. This rule is not essential and the protocol would also provide both safety and liveness if a party chose not to issue a support share in this case.
- *Optimizing small payloads.* For small payloads, instead of erasure coding the payload and dispersing fragments, the leader could just disperse the payload directly. A support share would also contain the payload as well. Alternatively, we could use an erasure code with different parameters that was more suitable for small payloads.

B.3 Simple variations on StableDispersedSimplex

The variations discussed in Appendix B.2 can be adapted to StableDispersedSimplex as well. Note that the variation in which support is withheld after complaining may be implemented so that after a party issues complaint share in an epoch, that party will not issue any more support shares in the epoch. This implementation will have the effect of dislodging the leader somewhat earlier. However, with this as well as with all of the other variations in Appendix B.2, the results for StableDispersedSimplex still hold.

C Comparison to other protocols

C.1 Simplex

As already mentioned above in Section 3.4.2, the optimistic proposal-commit latency (3δ) and the optimistic consecutive-proposal latency (2δ) of DispersedSimplex are the same as for Simplex. A proper comparison of the communication complexity of DispersedSimplex and Simplex is not really possible. This is because description of Simplex in [9] is a bit problematic. Taking the description of the protocol in Section 2.1 of [9] literally, in each slot, every party sends a copy of the *entire* blockchain, along with a support or complaint certificate for every slot from genesis, to every other party. This is clearly entirely impractical, and one must assume the authors of [9] mean this only in some figurative sense, although very little guidance is given as to what it should mean literally. Elsewhere (in particular in Section 3.4 of [9]) it is suggested that messages are much smaller (but without any details).

The variant of DispersedSimplex discussed in Appendix B.2 for small payloads, with no erasure coding, can be viewed as a fully specified, practical version of Simplex. The DispersedSimplex protocol itself then shows how to get even better communication complexity through erasure codes, but without increasing latency. Note that DispersedSimplex is optimistically responsive, just like Simplex.

C.2 HotStuff and HotStuff-2

We may also compare DispersedSimplex to HotStuff [27] and the recently proposed improvement HotStuff-2 [19].

C.2.1 Latency

HotStuff-2 has an optimistic proposal-commit latency of 5δ while HotStuff has a an optimistic proposal-commit latency of 7δ. Pipelined versions of these protocols can achieve an optimistic consecutive-proposal latency 2δ. Thus, (pipelined versions of) HotStuff and HotStuff-2 have the same optimistic consecutive-proposal latency of DispersedSimplex, but have worse optimistic proposal-commit latency (which is just 3δ for DispersedSimplex).

We note that HotStuff and HotStuff-2 are optimistically responsive, just like DispersedSimplex and Simplex.

C.2.2 Communication complexity

The reported communication complexity of HotStuff and HotStuff-2 is $O(n(\beta + \kappa + \lambda))$. Recall that β bounds the block size, κ the signature share/certificate size, and λ the hash size. For small blocks, specifically if $\beta \ll n(\kappa + \lambda \log n)$, this communication complexity is better than that of DispersedSimplex, which is $O(n\beta + n^2(\kappa + \lambda \log n))$, as we discussed above in Section 3.4.1. However, this reported communication cost does not actually take into account the cost of *reliable* block dissemination. In these protocols, the leader is (apparently) supposed to simply send its proposed block to each party – at least, that is what is written in [27].

This creates two problems. First, there is no mechanism specified that ensures that all honest parties obtain the payloads of committed blocks. Naive mechanisms in which parties simply poll other parties for missing blocks can easily degenerate into $O(n^2\beta)$ communication complexity: all corrupt parties could simply ask for a block from all honest parties. If information dispersal techniques are used to ensure data availability, this would again make the communication complexity quadratic in n. So at best, the communication complexity of these protocols is better only for small blocks and only assuming corrupt parties do not misbehave too much.

Second, if the description in [27] is taken literally, the communication load in HotStuff (and apparently HotStuff-2) is very *unbalanced*. This can create a communication bottleneck at the leader. Indeed, as demonstrated empirically in [20, 25], it seems that for systems with moderate network size (n up to a hundred or so) and large block sizes, taking care to disseminate blocks to all parties in a way that does not create a bottleneck at the leader is more important in practice than worrying about the quadratic dependence on n in the communication complexity. In contrast, as mentioned above in Section 3.4.1, the communication load of DispersedSimplex is *balanced*. That is, each party, *including the leader*, transmits roughly the same about of data over the network. Thus, while in HotStuff (and HotStuff-2), the leader has to transmit $O(n\beta)$ bytes across the network, in DispersedSimplex, the leader (and every party) transmits $O(\beta)$ bytes across the network.

C.2.3 Concrete estimates

It would be interesting to perform a careful empirical investigation to compare the real-world performance of DispersedSimplex and (pipelined) HotStuff/HotStuff-2 under various parameter settings. However, we can attempt to make a "back of the envelope" calculation, similar to what we did in Section 3.5. With the parameters we used there (1Gb/s network bandwidth and 100ms network latency), the propagation delay per slot would be the same, so about 200ms, and the computation delay would be less. As for the transmission delay, if the block size is β bytes, then in each slot the leader has to transmit a total of $n\beta$ bytes across the network. As a specific example, let us say $n \approx 100$, so the transmission delay would be about 800ms for every 1MB of block data. This is obviously much worse than the 25ms per 1MB of block data for DispersedSimplex. With these estimates, the best possible throughput that could be achieved is 1.25MB of block data per second. More concretely, suppose we set the block size to 1MB. So ignoring computation delay (which is just a few ms),

- the throughput is about 1MB per second (vs 18MB per second for DispersedSimplex, or 40MB per second for StableDispersedSimplex discussed in Section 4.2, or 80MB per second for the improved version of StableDispersedSimplex in the full version [22]),
- the optimistic consecutive-proposal latency is 1s (vs 440ms for DispersedSimplex, or 200ms for StableDispersedSimplex), and
- (for HotStuff-2) the optimistic proposal-commit latency is that plus about 300ms, so about 1.3s (vs 540ms for any of the variants of DispersedSimplex discussed here).

In the above calculations, we saw that for an unbalanced protocol like HotStuff (or PBFT), as n increases, the throughput should decrease, and the latency should increase, while in a balanced protocol like DispersedSimplex, throughput and latency should not depend very much on n. This type of behavior has been confirmed experimentally in papers such as [20, 25], although not for the exact protocols considered here. Also, while we focused on throughput and latency, there are other costs to consider – namely, the monetary (or other) costs associated with transmitting a certain *amount* of data. These costs are directly proportional to the overall communication complexity, and it is indeed true that erasure coding does inflate these costs by a factor of 3 (although this can be reduced to a factor of 1.5 as discussed in the full version [22]). Another factor to potentially consider is the fact that for a balanced protocol like DispersedSimplex, the rate at which each party is transmitting is fairy constant, while for protocols like HotStuff, it is very bursty.

C.3 ICC

The Simplex protocol bears a passing resemblance to the ICC protocols ICC in [7]. The main difference is that for the ICC protocols, if the leader for a slot v is perceived to fail, then instead of simply timing out, a (somewhat complicated) fail-over mechanism is triggered that will *eventually* add a block to the complete block tree for slot v that is proposed by a different party. Latency and communication costs in the optimistic setting for protocols ICC0 and ICC1 in [7] are very similar to that of Simplex. We note that protocol ICC2 in [7] employs information dispersal techniques to get better communication complexity, but at the expense extra latency. Thus, DispersedSimplex is both simpler and more efficient than that any of the ICC protocols.

C.4 DAG-based atomic broadcast protocols

Recently, there has been a flurry of papers on DAG-based atomic broadcast protocols [13, 10, 24, 23]. One of the attractions of these protocols is that, by design, they are leaderless and thereby avoid the bandwidth bottleneck that some leader-based protocols can exhibit. Indeed, as stated in [10]: "decoupling transaction dissemination from the critical path of consensus is the key to blockchain scalability". As mentioned above, the papers [20, 25] already demonstrated the importance of taking care to disseminate blocks to all parties in a way that does not create such a bottleneck. We also mentioned above that protocol ICC2 in [7] shows how to do this in a leader-based protocol, and we have shown in this paper how DispersedSimplex achieves this in a leader-based protocol with optimal proposal-commit latency. As shown in Section 4, a stable-leader variant of DispersedSimplex can achieve even better performance, and specifically, when the leader is honest and the network is synchronous, all parties will essentially *fully utilize all available network bandwidth.* Thus, it is not entirely clear to us that the leader-bottleneck problem exhibited by some earlier leader-based protocols is a valid reason to abandon leader-based protocols entirely, especially since leader-based protocols (such as DispersedSimplex) still exhibit superior (and essentially optimal) latency characteristics. Moreover, it is also not entirely clear to us that "decoupling transaction dissemination from the critical path of consensus" is an inherently good idea: while such a decoupling may be good from a software engineering point of view, as we demonstrate with DispersedSimplex, it is precisely by *tightly coupling* dissemination with consensus that *we can fully utilize network bandwidth without sacrificing optimal latency*, using a quite simple and elegant protocol.

There are many metrics on which consensus protocols may be compared. While DAG-based consensus protocols may well be superior on some metrics, it does not appear (based on our analysis) that the core metrics of common-case throughput and latency are among them.

Near-Linear Time Dispersion of Mobile Agents

Yuichi Sudo
Hosei University, Tokyo, Japan

Masahiro Shibata
Kyushu Institute of Technology, Fukuoka, Japan

Junya Nakamura
Toyohashi University of Technology, Aichi, Japan

Yonghwan Kim
Nagoya Institute of Technology, Aichi, Japan

Toshimitsu Masuzawa
Osaka University, Osaka, Japan

Abstract

Consider that there are $k \leq n$ agents in a simple, connected, and undirected graph $G = (V, E)$ with n nodes and m edges. The goal of the dispersion problem is to move these k agents to mutually distinct nodes. Agents can communicate only when they are at the same node, and no other communication means, such as whiteboards, are available. We assume that the agents operate synchronously. We consider two scenarios: when all agents are initially located at a single node (rooted setting) and when they are initially distributed over one or more nodes (general setting). Kshemkalyani and Sharma presented a dispersion algorithm for the general setting, which uses $O(m_k)$ time and $\log(k + \Delta)$ bits of memory per agent [OPODIS 2021], where m_k is the maximum number of edges in any induced subgraph of G with k nodes, and Δ is the maximum degree of G. This algorithm is currently the fastest in the literature, as no $o(m_k)$-time algorithm has been discovered, even for the rooted setting. In this paper, we present significantly faster algorithms for both the rooted and the general settings. First, we present an algorithm for the rooted setting that solves the dispersion problem in $O(k \log \min(k, \Delta)) = O(k \log k)$ time using $O(\log(k + \Delta))$ bits of memory per agent. Next, we propose an algorithm for the general setting that achieves dispersion in $O(k \log k \cdot \log \min(k, \Delta)) = O(k \log^2 k)$ time using $O(\log(k + \Delta))$ bits. Finally, for the rooted setting, we give a time-optimal (i.e., $O(k)$-time) algorithm with $O(\Delta + \log k)$ bits of space per agent. All algorithms presented in this paper work only in the synchronous setting, while several algorithms in the literature, including the one given by Kshemkalyani and Sharma at OPODIS 2021, work in the asynchronous setting.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases mobile agents, autonomous robots, dispersion

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.38

Related Version *Full Version*: https://doi.org/10.48550/arXiv.2310.04376 [19]

Funding JSPS KAKENHI 20KK0232.
Yuichi Sudo: JSPS KAKENHI 20H04140, JST FOREST Program JPMJFR226U.
Masahiro Shibata: JSPS KAKENHI 21K17706 and 23K28037.
Junya Nakamura: JSPS KAKENHI 22K11971.

1 Introduction

In this paper, we focus on the dispersion problem involving mobile entities, referred to as mobile agents, or simply, *agents*. At the start of an execution, k agents are arbitrarily positioned at nodes of an undirected graph $G = (V, E)$ with n nodes and m edges. The objective is to ensure that all agents are located at mutually distinct nodes. This problem

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 38; pp. 38:1–38:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Dispersion of mobile agents on an arbitrary graph ($\tau = \min(k, \Delta)$) The algorithm of [7] needs to know an asymptotically tight upper bound on m_k in advance. (Since $m_k \le \min(m, k\Delta, \binom{k}{2})$), knowing tight upper bounds on m, k, and Δ is sufficient, but it increases the running time to $O(\min(m, k\Delta, \binom{k}{2}) \log k)$.) The $\log k$ term can be eliminated from the space complexities marked with daggers (†) if you choose to disregard the memory space required for each agent to store its own identifier. For example, the proposed algorithm mentioned in Theorem 8 requires only $O(\log \Delta)$ bits per agent. The space complexity of the first algorithm given by [6], which is marked with a double dagger (‡), can be decreased to $O(k \log \Delta)$ if we assume that the number of possible agent-identifiers is $O(k)$. All algorithms listed in this table are deterministic.

	Memory per agent	Time	General/Rooted	Async./Sync.
[2]	$O(\log(k + \Delta))$ †	$O(m_k)$	rooted	async.
[10]	$O(D + \Delta \log k)$	$O(D\Delta(D + \Delta))$	rooted	async.
Theorem 8	$O(\log(k + \Delta))$ †	$O(k \log \tau)$	rooted	sync.
Theorem 12	$O(\Delta + \log k)$ †	$O(k)$	rooted	sync.
[6]	$O(k \log(k + \Delta))$ ‡	$O(m_k)$	general	async.
[6]	$O(D \log \Delta + \log k)$ †	$O(\Delta^D)$	general	async.
[6]	$O(\log(k + \Delta))$	$O(m_k \cdot k)$	general	async.
[7]	$O(\log(k + \Delta))$	$O(m_k \log k)$	general	sync.
[16]	$O(\log(k + \Delta))$	$O(m_k \log k)$	general	sync.
[11]	$O(\log(k + \Delta))$	$O(m_k)$	general	async.
Theorem 10	$O(\log(k + \Delta))$	$O(k \log^2 k)$	general	sync.
Lower bound	any	$\Omega(k)$	any	any

was originally proposed by Augustine and Moses Jr. [2] in 2018. A particularly intriguing aspect of this problem is the unique computation model. Unlike many other models involving mobile agents on graphs, we do not have access to node identifiers, nor can we use local memory at each node. In this setting, an agent cannot retrieve or store any information from or on a node when it visits. However, each of the k agents possesses a unique identifier and can communicate with each other when they are at the same node in the graph. The agents must collaboratively solve the dispersion problem through this direct communication.

Several algorithms have been introduced in the literature to solve the dispersion problem. This problem has been examined in two different contexts within the literature: the *rooted setting* and the *general setting*. In the rooted setting, all k agents initially reside at a single node. On the other hand, the general setting imposes no restrictions on the initial placement of the k agents. For any $i \in [1, n]$, we define m_i as the maximum number of edges in any i-node induced subgraph of G. The parameter m_k, where k is the number of agents, serves as an upper bound on the number of edges connecting two nodes, each hosting at least one agent, in any configuration. Consequently, m_k frequently appears in the time complexities of dispersion algorithms. This is because (i) solving the dispersion problem essentially requires finding k distinct nodes, and (ii) the simple depth-first search (DFS), employed as a submodule by many dispersion algorithms, needs to explore m_k edges to find k nodes.

Table 1 provides a summary of various dispersion algorithms found in the literature, all designed for arbitrary graphs. Here, Δ and D are the maximum degree and the diameter of a graph, respectively, and $\tau = \min(k, \Delta)$. Augustine and Moses Jr. [2] introduced a simple algorithm, based on depth-first search (DFS), for the rooted setting. This algorithm solves the dispersion problem in $O(m_k)$ time using $O(\log \Delta)$ bits of space per agent. Kshemkalyani and Ali [6] provides two algorithms that accomplish dispersion in the general setting: an $O(m_k)$-time and $O(k \log(k + \Delta))$-space algorithm, and an $O(m_k \cdot k)$-time and $O(\log(k + \Delta))$-

space algorithm, offering a trade-off between time and space. The first is faster but needs more space, while the second is slower but more memory-efficient. Kshemkalyani, Molla, and Sharma [7] found a middle ground with an algorithm that runs in $O(m_k \log k)$ time and uses $O(\log(k + \Delta))$ bits of each agent's memory. This algorithm, however, requires a priori global knowledge, asymptotically tight upper bounds on m_k, to attain its time upper bound. Shintaku, Sudo, Kakugawa, and Masuzawa [16] managed to eliminate this requirement for global knowledge. More recently, Kshemkalyani and Sharma [11] removed the $\log k$ factor from the running time. This algorithm also works in an asynchronous setting, meaning the agents do not need to share a common clock. Any dispersion algorithm requires at least $\Omega(k)$ time, which is almost trivial, but we will provide a proof for completeness in this paper. No other lower bounds on the time complexity of dispersion have been established in the literature. Thus, there is still a significant gap between the best known upper bound $O(m_k)$ and this lower bound of $\Omega(k)$ because $m_k = \Theta(k^2)$ holds in many graph classes. Note that $m_k = \Theta(k^2)$ may hold even in a sparse graph when $k = O(\sqrt{n})$.

All the algorithms mentioned above are based on DFS. However, a few algorithms [6, 10] are designed based on BFS (breadth-first search) and exhibit different performance characteristics. Notably, their upper bounds on running time do not depend on the number of agents k, but depend on diameter D and the maximum degree Δ of a graph.

▶ Note 1 (Space Complexity). Conforming to the convention in the studies of mobile agents [5], this paper, including Table 1, evaluates the space complexity of an algorithm as the maximum size of *persistent memory* needed by an agent during its execution. Persistent memory refers to the information an agent carries when it moves from one node to another and does not include the *working memory* used for local computations at nodes. This persistent memory includes the space required to store its own identifier. Since the k agents are labeled with unique identifiers, every algorithm requires $O(\log k)$ bits per agent.

▶ Note 2 (Parameter m_k). The parameter m_k is introduced in this paper and has not been previously utilized in the literature. Traditionally, the running times of DFS-based algorithms are represented using the parameter $\min(m, k\Delta)$ or $\min(m, k\Delta, \binom{k}{2})$, which are always greater than or equal to m_k. The parameter m_k may be better to represent them because there are some graph classes where $m_k = o(k^2)$ while $\min(m, k\Delta, \binom{k}{2}) = \Omega(k^2)$.

1.1 Our Contribution

In this paper, we drastically reduce the gap between the upper bound $O(m_k)$ and the lower bound $\Omega(k)$ mentioned earlier. As previously noted, $m_k = \Theta(k^2)$ may hold even in sparse graphs, making this gap significant. Let $\tau = \min(k, \Delta)$. We present two algorithms: one for the rooted setting that achieves dispersion in $O(k \log \tau) = O(k \log k)$ time using $O(\log \Delta)$ bits, and the other for the general setting that achieves dispersion in $O(k \log k \cdot \log \tau) = O(k \log^2 k)$ time using $O(\log(k + \Delta))$ bits. The upper bounds obtained here match the lower bounds in both the rooted and the general settings when ignoring poly-logarithmic factors.

To achieve this upper bound, we introduce a new technique. Like many existing algorithms, our algorithms are based on Depth-First Search (DFS). That is, we let agents run DFS on a graph and place or *settle* an agent at each unvisited node they find. Each time unsettled agents find an unvisited node v, one of the agents settles at v, and the others try to find an unvisited neighbor of v. If such a neighbor exists, they move to it. If no such neighbor exists, they go back to the parent of v in the DFS tree. To find an unvisited neighbor, all DFS-based dispersion algorithms in the literature make the unsettled agents visit those neighbors sequentially, i.e., one by one. This process obviously requires $\Omega(\tau)$ time. We break this barrier and find an unvisited neighbor of v in $O(\log \tau) = O(\log k)$ time, with the help of the agents already settled at neighbors of the current location v.

Our goal here is to find any one unvisited neighbor of v if it exists, not to find all of them. Consider the case where there are only two agents a and b at v, a is settled at v, and b is still an unsettled agent. Agent b visits a neighbor of v and if b finds a settled agent at that node, b brings that agent to v. Consequently, there are two agents on v, excluding a, so we can use these two to visit two neighbors of v in parallel. Again, if there are settled agents on both nodes, those agents will be brought to v. Importantly, the number of agents at v, excluding a, doubles each time this process is repeated until an unvisited neighbor is found. Therefore, over time, we can check neighbors of v in parallel with an exponentially increasing number of agents. As a result, we can finish this search or *probing* process in $O(\log \tau)$ time. Thereafter, we allow the helping agents we brought to v to return to their original nodes, or their *homes*. Since we perform the probing process only $O(k)$ times in total throughout DFS, a simple analysis shows that dispersion can be achieved in $O(k \log \tau)$ time in the rooted setting. We call the resulting DFS the *HEO (Helping Each Other)-DFS* in this paper.

In the general setting, like in existing studies, we conduct multiple DFSs in parallel, each starting from a different node. While the DFS performed in existing research requires $\Theta(m_k)$ time, we use HEO-DFS, thus each DFS completes in $O(k \log \tau)$ time. Thus, at first glance, it seems that dispersion can be achieved in $O(k \log \tau)$ time. However, this analysis does not work so simply because each DFS interferes with each other. Our proposed algorithm employs the method devised by Shintaku et al. [16] to efficiently merge multiple DFSs and run HEO-DFSs in parallel with this method. The merge process incurs an $O(\log k)$ overhead, so we solve the dispersion problem in $O(k(\log k) \cdot (\log \tau)) = O(k \log^2 k)$ time.

It might seem that the overhead can be eliminated by using the DFS parallelization method proposed by Kshemkalyani and Sharma [11], instead of the method of Shintaku et al. [16]. However, this is not the case because our HEO-DFS is not compatible with the parallelization method of Kshemkalyani and Sharma. Specifically, their method entails a process such that one DFS absorbs another when multiple DFSs collide. During this process, it is necessary to gather the agents in the absorbed side to a single node, which requires $\Theta(m_k)$ time. Our speed-up idea effectively works for finding an unvisited neighbor, but it does not work for the acceleration of gathering agents dispersed on multiple nodes. Therefore, it is unlikely that our HEO-DFS can be combined with the method of Kshemkalyani and Sharma.

The two algorithms mentioned above are nearly time-optimal, i.e., requiring $O(k \cdot \log^c k)$ time for some constant c. We also demonstrate that in the rooted setting, a time-optimal algorithm based on the HEO-DFS can be achieved if significantly more space is available, specifically $O(\Delta)$ bits per agent.

To the best of our knowledge, HEO-DFS is a novel approach, and no similar techniques have been used in the literature concerning mobile agents and mobile robots. While we demonstrate that HEO-DFS significantly reduces the running time of dispersion algorithms, this technique may also prove useful for addressing other fundamental problems such as exploration and gathering.

A drawback of our HEO-DFS is that it fundamentally requires a synchronous model, even in the rooted setting; i.e., it does not function in an asynchronous model. In HEO-DFS, we attempt to find an unvisited neighbor of the current location with the help of agents settled on other neighbors. These agents must return to their homes once the probing process is completed. In an asynchronous model, unsettled agents (and/or helping agents) may visit those homes before their owners return, disrupting the consistent behavior of HEO-DFS. Therefore, the algorithm provided by Kshemkalyani and Sharma [11] remains the fastest for the asynchronous model. It is still an open question whether there exists a $o(k^2)$-time algorithm that accommodates asynchronicity.

▶ Note 3 (Termination). In this paper, we do not explicitly mention how the agents terminate the execution of a given algorithm. In many cases, termination is straightforward without any additional assumptions in the rooted setting, while in the general setting, additional assumptions are required. Specifically, in the general setting, all algorithms listed in Table 1, except for the $O(\Delta^d)$-time algorithm presented in [8] [1], require both a synchronous setting and global knowledge such as (asymptotically tight upper bounds on) m_k and k. With these assumptions, the agents can easily terminate simultaneously after a sufficiently large number of steps, e.g., $\Theta(k \log^2 k)$ steps in our algorithm for the general setting. Thus, when termination is required, our general setting algorithm no longer exhibits disadvantages compared to existing algorithms: all existing algorithms, except for the $O(\Delta^D)$-time one [8], also require a synchronous setting (and some global knowledge).

For completeness, we present how the agents terminate in our algorithms for the rooted setting, which is almost trivial, in the arXiv version [19].

1.2 Further Related Work

The dispersion problem has been studied not only for arbitrary undirected graphs but also for graphs with restricted topologies such as trees [2], grids [7, 9], and dynamic rings [1]. Additionally, several studies have explored randomized algorithms to minimize the space complexity of dispersion [13, 4], and others have focused on fault-tolerant dispersion [12, 3]. Kshemkalyani et al. [10] introduced the *global communication model*, where all agents can communicate with each other regardless of their locations. In contrast, the standard model, where only the agents co-located at the same node can communicate with each other, is sometimes referred to as the *local communication model.* All algorithms listed in Table 1 assume the local communication model and are deterministic.

Exploration by a single mobile agent is closely related to the dispersion problem. The exploration problem requires an agent to visit all nodes of a graph. Many studies have addressed the exploration problem, and numerous efficient algorithms, both in terms of time and space, have been presented in the literature [15, 14, 17, 18]. In contrast to exploration, the dispersion problem only requires finding k nodes, and we can use k agents to achieve this. Our HEO-DFS take advantage of these differences to solve the dispersion problem efficiently.

2 Preliminaries

Let $G = (V, E)$ be any simple, undirected, and connected graph. Let $n = |V|$ and $m = |E|$. We denote the set of *neighbors* of node $v \in V$ by $N(v) = \{u \in V \mid \{u, v\} \in E\}$ and the degree of a node v by $\delta_v = |N(v)|$. Let $\Delta = \max_{v \in V} \delta_v$, i.e., Δ is the maximum degree of G. The nodes are anonymous, i.e., they do not have unique identifiers. However, the edges incident to a node v are locally labeled at v so that an agent located at v can distinguish those edges. Specifically, those edges have distinct labels $0, 1, \dots, \delta_v - 1$ at node v. We call these local labels *port numbers.* We denote the port number assigned at v for edge $\{v, u\}$ by $p_v(u)$. Each edge $\{v, u\}$ has two endpoints, thus has labels $p_u(v)$ and $p_v(u)$. Note that these labels are independent, i.e., $p_u(v) \neq p_v(u)$ may hold. For any $v \in V$, we define $N(v, i)$ as the node $u \in N(v)$ such that $p_v(u) = i$. For simplicity, we define $N(v, \perp) = v$ for all $v \in V$.

[1] However, in this algorithm, the agents do not terminate simultaneously, and they require the ability to detect whether or not there is a terminated agent at the current location.

We consider that k agents exist in graph G, where $k \leq n$. The set of all agents is denoted by A. Each agent is always located at some node in G, i.e., the move of an agent is *atomic* and an agent is never located at an edge at any time step (or just *step*). The agents have unique identifiers, i.e., each agent a has a positive integer as its identifier $a.\mathtt{ID}$ such that $a.\mathtt{ID} \neq b.\mathtt{ID}$ for any $b \in A \setminus \{a\}$. The agents know a common upper bound $\mathrm{id}_{\max} \geq \max_{a \in A} a.\mathtt{ID}$ such that $\mathrm{id}_{\max} = \mathrm{poly}(k)$, thus the agents can store the identifier of any agent on $O(\log k)$ space. Each agent has a read-only variable $a.\mathtt{pin} \in \{0, 1, \dots, \Delta - 1\} \cup \{\bot\}$. At time step 0, $a.\mathtt{pin} = \bot$ holds. For any $t \geq 1$, if a moves from u to v at step $t - 1$, $a.\mathtt{pin}$ is set to $p_v(u)$ (or the port of v incoming from u) at the beginning of step t. If a does not move at step $t - 1$, $a.\mathtt{pin}$ is set to $\bot$. We call the value of $a.\mathtt{pin}$ the incoming port of a. The values of all variables in agent a, excluding its identifier $a.\mathtt{ID}$ and special variables $a.\mathtt{pin}, a.\mathtt{pout}$, constitute the state of a. (We will see what is $a.\mathtt{pout}$ later.)

The agents are synchronous and are given a common algorithm $\mathcal{A}$. An algorithm $\mathcal{A}$ must specify the initial state s_{init} of agents. All agents are in state s_{init} at time step 0. Let $A(v, t) \subseteq A$ denote the set of agents located at node v at time step $t \geq 0$. At each time step $t \geq 0$, each agent $a \in A(v, t)$ is given the following information as the inputs: (i) the degree of v, (ii) its identifier $a.\mathtt{ID}$, and (iii) a sequence of triples $((b.\mathtt{ID}, s_b, b.\mathtt{pin}))_{b \in A(v,t)}$, where s_b is the current state of b. Note that each $a \in A(v, t)$ can obtain its current state s_a and $a.\mathtt{pin}$ from the sequence of triples since a is given its ID as the second information. Then, it updates the variables in its memory space in step t, including a variable $a.\mathtt{pout} \in \{\bot, 0, 1, \dots, \delta_v - 1\}$, according to algorithm $\mathcal{A}$. Finally, each agent $a \in A(v, t)$ moves to node $N(v, a.\mathtt{pout})$. Since we defined $N(v, \bot) = v$ above, agent a with $a.\mathtt{pout} = \bot$ stays in v in step t.

A node does not have any local memory accessible by the agents. Thus, the agents can coordinate only by communicating with the co-located agents. No agents are given any global knowledge such as m, Δ, k, and m_k in advance.

A function $C : A \to \mathcal{M}_{\mathcal{A}} \times V \times \{\bot, 0, 1, \dots, \Delta - 1\}$ is called a global state of the network or a *configuration* if $C(a) = (s, v, q)$ yields $q = \bot$ or $q < \delta_v$ for any $a \in A$, where $\mathcal{M}_{\mathcal{A}}$ is the (possibly infinite) set of all agent-states. A configuration specifies the state, location, and incoming port of each $a \in A$. In this paper, we consider only deterministic algorithms. Thus, if the network is in a configuration C at a time step t, a configuration C' in the next step $t + 1$ is uniquely determined. We denote this configuration C' by $\mathit{next}_{\mathcal{A}}(C)$. The execution $\Xi_{\mathcal{A}}(C_0)$ of algorithm $\mathcal{A}$ starting from a configuration C_0 is defined as an infinite sequence $C_0, C_1, \dots$ of configurations such that $C_{t+1} = \mathit{next}_{\mathcal{A}}(C_t)$ for all $t = 0, 1, \dots$. We say that a configuration C_0 is *initial* if the states of all agents are s_{init} and the incoming ports of all agents are $\bot$ in C_0. Moreover, in the rooted setting, we restrict the initial configurations to those where all agents are located at a single node.

▶ **Definition 4** (Dispersion Problem). *A configuration C of an algorithm $\mathcal{A}$ is called* legitimate *if (i) all agents in A are located in different nodes in C, and (ii) no agent changes its location in execution $\Xi_{\mathcal{A}}(C)$. We say that $\mathcal{A}$ solves the dispersion problem if execution $\Xi_{\mathcal{A}}(C_0)$ reaches a legitimate configuration for any initial configuration C_0.*

We evaluate the *time complexity* or *running time* of algorithm $\mathcal{A}$ as the maximum number of steps until $\Xi_{\mathcal{A}}(C_0)$ reaches a legitimate configuration, where the maximum is taken over all initial configurations C_0. Let $\mathcal{M}'_{\mathcal{A}} \subseteq \mathcal{M}_{\mathcal{A}}$ be the set of all agent-states that can appear in any possible execution of $\mathcal{A}$ starting from any initial configuration. We evaluate the *space complexity* or *memory space* of algorithm $\mathcal{A}$ as $\log_2 |\mathcal{M}'_{\mathcal{A}}| + \log_2 \mathrm{id}_{\max}$, i.e., the maximum number of bits required to represent an agent-state that may appear in those executions, plus the number of bits required for each agent to store its own identifier. This implies that we exclude the size of the working memory used for deciding the destination and updating states, as well as the space for storing input information, except for the agent's own identifier.

Throughout this paper, we denote by $[i,j]$ the set of integers $\{i, i+1, ..., j\}$. We have $[i,j] = \emptyset$ when $j < i$. When the base of a logarithm is not specified, it is assumed to be 2. We frequently use $\tau = \min(k, \Delta)$. We define $\nu(a,t)$ as the node where agent a resides at time step t. We also omit time step t from any function in the form $f(*, t)$ and just write $f(*)$ if t is clear from the context. For example, we just write $A(v)$ and $\nu(a)$ instead of $A(v,t)$ and $\nu(a,t)$.

We have the following remark considering the fact that G can be a simple path.

▶ Remark 5. For any dispersion algorithm $\mathcal{A}$, there exists a graph G such that an execution of $\mathcal{A}$ requires $\Omega(k)$ time steps to achieve dispersion on both the rooted and the general settings.

In the two algorithms we present in this paper, **RootedDisp** and **GeneralDisp**, each agent maintains a variable $a.\mathtt{settled} \in \{\bot, \top\}$. We say that an agent a is a *settler* when $a.\mathtt{settled} = \top$, and an *explorer* otherwise. All agents are explorers initially. Once an explorer becomes a settler, it never becomes an explorer again. Let t be the time at which an agent a becomes a settler. Thereafter, we call the location of a at that time, i.e., $\nu(a,t)$, the *home* of a. Formally, a's home at $t' \geq 0$, denoted by $\xi(a,t')$, is defined as $\xi(a,t') = \bot$ if $t' < t$ and $\xi(a,t') = \nu(a,t)$ otherwise. It is worth mentioning that a settler may temporarily leave its home. Hence $\xi(a,t') = \nu(a,t')$ may not always hold even after a becomes a settler, i.e., even if $t' \geq t$. However, by definition, no agent changes its home. We say that an agent a *settles* when it becomes a settler.

When a node u is a home of an agent at time step t, we call this agent the settler of u and denote it as $\psi(u,t)$. Formally, if there exists an agent a such that $\xi(a,t) = u$, then $\psi(u,t) = a$; otherwise, $\psi(u,t) = \bot$. This function ψ is well defined for the two presentented algorithms because they ensure that no two agents share a common home. We say that a node u is *unsettled* at time step t if $\psi(u,t) = \bot$, and *settled* otherwise.

3 Rooted Dispersion

In this section, we present an algorithm, **RootedDisp**, that solves the dispersion problem in the rooted setting. That is, it operates under the assumption that all agents are initially located at a single node $s \in V$. This algorithm straightforwardly implements the strategy of the HEO-DFS, which we presented in Section 1. The time and space complexities of this algorithm are $O(k \log \tau)$ steps and $O(\log(k + \Delta))$ bits, respectively.

In an execution of Algorithm **RootedDisp**, the agent with the largest ID, denoted as $a_{\max}$, serves as the leader. Note that every agent can easily determine whether it is $a_{\max}$ or not at time step 0 by comparing the IDs of all agents. Then, $a_{\max}$ conducts a depth-first search (DFS), while the other agents move with the leader and one of them settles at an unsettled node when they visit it. If $a_{\max}$ encounters an unsettled node without any accompanying agents, $a_{\max}$ settles itself on that node, achieving dispersion. During a DFS, $a_{\max}$ must determine (i) whether there is an unsettled neighbor of the current location, and (ii) if so, which neighbor is unsettled. To make this decision, all DFS-based algorithms in the literature have $a_{\max}$ visit neighbors one by one until it finds an unsettled node, which clearly requires $\Omega(\tau)$ steps. **RootedDisp**, in contrast, makes this decision in $O(\log \tau)$ steps with the help of the agents that have already settled on the neighbors of the current location.

The pseudocode for Algorithm **RootedDisp** is shown in Algorithm 1. This pseudocode consists of two parts: the main function (lines 1–12) and the function $\mathtt{Probe}()$ (lines 13–23). As mentioned in the previous section, every agent a maintains a variable $a.\mathtt{settled} \in \{\bot, \top\}$, which decides whether a is an explorer or a settler. In addition, the settler $\psi(w)$ of a node w

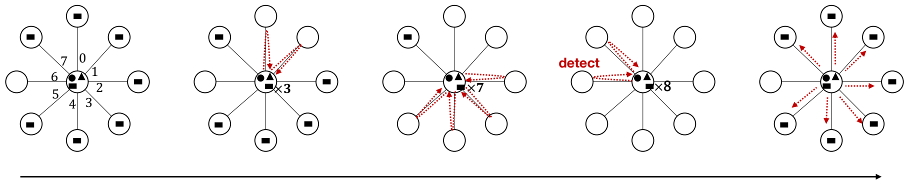


Figure 1 The behavior of the agents when the leader $a_{\max}$ invokes `Probe`() at the center node w in **RootedDisp**. A black circle, triangle, and rectangle represent a leader ($a_{\max}$), a non-leader explorer, and a settler, respectively. The integers in the leftmost figure represents port numbers. In every two time steps, the number of agents on w excluding $\psi(w)$ doubles (i.e., 2 → 4 → 8) until some agent detects an unsettled neighbor of w. After that, $a_{\max}$ lets the helping settlers go back to their homes.

maintains two variables, $\psi(w).\texttt{parent}, \psi(w).\texttt{next} \in [0, \delta_w - 1] \cup \perp$ for the main function. As we will see later, the following are guaranteed each time $a_{\max}$ invokes `Probe`() at node w:

- If there exists an unsettled node in $N(w)$, the corresponding port number will be stored in $\psi(w).\texttt{next}$. More precisely, an integer i such that $N(w, i) = u$ and $\psi(u) = \perp$ is assigned to $\psi(w).\texttt{next}$.
- If all neighbors are settled, $\psi(w).\texttt{next}$ will be set to $\perp$.
- `Probe`() will return in $O(\log \tau)$ time.

The main function performs a depth-first search using function `Probe`() to achieve dispersion. At the beginning of the execution, all agents are located at the same node s. Initially, the agent with the smallest ID settles at node s, and $\psi(s).\texttt{parent}$ is set to $\perp$ (lines 1–2). Then, as long as there are unsettled nodes in $N(\nu(a_{\max}))$, all explorers move to one of those nodes together (lines 7–8). We call this kind of movements *forward moves*. After each forward move from a node w to u, the agent with the smallest ID among $A(u)$ settles on u, and $\psi(u).\texttt{parent}$ is set to i with $N(u, i) = w$ (lines 9–10). For any node $u \in V$, if $\psi(u).\texttt{parent} \neq \perp$, we say that $w = N(u, \psi(u).\texttt{parent})$ is a *parent* of u. By line 9–10, each of the nodes except for the starting node s will have its parent as soon as it becomes settled. When the current location has no unsettled neighbors, all explorers move to the *parent* of the current location (lines 11–12). We call this kind of movements *backward moves* or *retreats*. Finally, $a_{\max}$ terminates when it settles (line 3).

Since the number of agents is k, the DFS-traversal stops after $a_{\max}$ makes a forward move $k-1$ times. The agent $a_{\max}$ makes a backward move at most once from any node. Therefore, excluding the execution time of `Probe`(), the execution of the main function completes in $O(k)$ time. Furthermore, the function `Probe`() is invoked at most $2(k-1)$ times, once after each forward move and once after each backward move. Since a single invocation of `Probe`() requires $O(\log \tau)$ time, the overall execution time of **RootedDisp** can be bounded by $O(k \log \tau)$ time.

Let us describe the behavior of the function `Probe`(), assuming that it is invoked on node w at time step t. Figure 1 may help the readers to understand the behavior. In the execution of `Probe`(), the leader $a_{\max}$ employs the explorers present on w and (a portion of) the settlers at $N(w)$ to search for an unsettled node in $N(w)$. We implement this process with a variable $\psi(w).\texttt{checked} \in [-1, \delta_w - 1]$ for the settler $\psi(w)$. Specifically, explorers at node w verify whether the neighbors of w are unsettled or not in the order of port numbers and store the most recently checked port number in $\psi(w).\texttt{checked}$. Consequently, $\psi(w).\texttt{checked} = \ell$ implies that the neighbors $N(w, 0), N(w, 1), \ldots, N(w, \ell)$ are settled. Let $x = |A(w, t) \setminus \{\psi(w)\}|$, i.e., there are x agents at w when `Probe`() is invoked, excluding $\psi(w)$.

Algorithm 1 RootedDisp.

```
1  $b$.settled $\leftarrow \top$, where $b$ is the agent with the smallest ID in $A(s)$
                                        // $b$ settles at the starting node $s$
2  $b$.parent $\leftarrow \bot$
3  while $a_{\max}$.settled $= \bot$ do
4      Probe()
5      Let $w = \nu(a_{\max})$
6      if $\psi(w)$.next $\neq \bot$ then
7          Let $u = N(w, \psi(w).\texttt{next})$      // $u$ is an unsetteled node here
8          All explorers in $A(w)$ go to $u$
9          $b'$.settled $\leftarrow \top$, where $b'$ is the agent with the smallest ID in $A(u)$
10         $b'$.parent $\leftarrow a_{\max}$.pin
11     else
12         All explorers in $A(w)$ go back to node $N(w, \psi(w).\texttt{parent})$.

13 function Probe():
14     Let $w = \nu(a_{\max})$.
15     $(\psi(w).\texttt{next}, \psi(w).\texttt{checked}) \leftarrow (\bot, -1)$
16     while $\psi(w)$.checked $\neq \delta_w - 1$ do
17         Let $a_1, a_2, \dots, a_x$ be the agents in $A(w) \setminus \{\psi(w)\}$, and let
             $\Delta' = \min(x, \delta_w - 1 - \psi(w).\texttt{checked})$. For each $i = 1, 2, \dots, \Delta'$, assign $a_i$ to
             the neighboring node $u_i = N(w, i + \psi(w).\texttt{checked})$, and let $a_i$ make a round
             trip between $w$ and $u_i$. In other words, make $a_i$ move in the order
             $w \rightarrow u_i \rightarrow w$. If $a_i$ finds a settler at $u_i$, it will bring the settler $\psi(u_i)$ back to
             $w$.
18         if there exists $a_i$ that did not bring $\psi(u_i)$ back to $w$ then
19             $\psi(w)$.next $\leftarrow i + \psi(w)$.checked       // $u_i$ must be unsettled
20             Break the while loop.
21         else
22             $\psi(w)$.checked $\leftarrow \psi(w)$.checked $+ \Delta'$
23     Let all settlers except for $\psi(w)$ go back to their homes.
```

In the first iteration of the while loop (lines 16–22), the $\min(x, \delta_w)$ agents concurrently visit $\min(x, \delta_w)$ neighbors and then return to w (lines 17–18). This entire process takes exactly two time steps. These agents bring back all the settlers, at most one for each neighbor, they find. If there is an agent that does not find a settler, then the node visited by that agent must be unsettled. In such a case, the port used by one of these agents is stored in $\psi(w)$.next, and the while loop terminates (lines 20–21).[2] If all x agents bring back one agent each, then there are $2x$ agents on w, excluding $\psi(w)$. In the second iteration of the while loop, these $2x$ agents visit the next $2x$ neighbors and search for unsettled neighbors in a similar way. As long as no unsettled neighbors are discovered, the number of agents on w, excluding $\psi(w)$,

[2] For simplicity, we reset $\psi(w)$.checked to -1 each time we invoke Probe() at w, so we do not use the information about which ports were already checked in the past invocation of Probe(). As a result, the value of $\psi(w)$.next computed by Probe does not have to be the minimum port leading to an unsettled neighbor of w.

doubles with each iteration of the while loop. Since there are at most $\tau = \min(k, \Delta)$ settled nodes in $N(w)$, after running the while loop at most $O(\log(\tau/x)) = O(\log \tau)$ times, either an unsettled node will be found, or the search will be concluded without finding any unsettled nodes. In the latter case, since $\psi(w)$.`next` is initialized to $\perp$ when `Probe`() is called (line 15), $\psi(w)$.`next` $= \perp$ will also be valid at the end of the while loop, allowing $a_{\max}$ to verify that all neighbors of w are settled. After the while loop ends, the settlers brought back to w return to their homes (line 23). This process of "returning to their homes" requires the agents to remember the port number leading to their home from w. However, we exclude this process from the pseudocode because it can be implemented in a straightforward manner, and it requires only $O(\log \Delta)$ bits of each agent's memory. In conclusion, we have the following lemma.

▶ **Lemma 6.** *Each time* `Probe`() *is invoked on node* $w \in V$*,* `Probe`() *finishes in* $O(\log \tau)$ *time. At the end of* `Probe`()*, it is guaranteed that: (i) if there exists an unsettled node in* $N(w)$*, then* $N(w, \psi(w).\texttt{next})$ *is unsettled, and (ii) if there are no unsettled nodes in* $N(w)$*, then* $\psi(w)$.`next` $= \perp$ *holds true.*

▶ **Lemma 7.** *Each agent requires* $O(\log(k + \Delta))$ *bits of memory to execute* **RootedDisp***.*

Proof. In this algorithm, an agent handles several $O(\log \Delta)$-bit variable, `next`, `checked`, `parent`, as well as the port number that the settler $\psi(u)$ needs to remember in order to return to node u from node w at line 23 after coming at line 17. Every other variable can be stored in a constant space. Therefore, the space complexity is $O(\log(k + \Delta))$ bits, adding the memory space to store the agent's identifier. ◀

▶ **Theorem 8.** *In the rooted setting, algorithm* **RootedDisp** *solves the dispersion problem within* $O(k \log \tau)$ *time using* $O(\log(k + \Delta))$ *bits of space per agent.*

Proof. As long as there is an unsettled neighbor of the current location, $a_{\max}$ makes a forward move to one of those nodes. If there is no such neighbor, $a_{\max}$ makes a backward move to the parent node of the current location. Since the graph is connected, this DFS-traversal clearly visits k nodes with exactly $k - 1$ forward moves and at most $k - 1$ backward moves. Thus, the number of calls to `Probe`() is at most $2(k - 1)$ times. By Lemma 6, the execution of **RootedDisp** achieves dispersion within $O(k \log \tau)$ time. ◀

4 General Dispersion

4.1 Overview

In this section, we present an algorithm **GeneralDisp** that solves the dispersion problem in $O(k \log \tau \cdot \log k) = O(k \log^2 k)$ time, using $O(\log(k + \Delta))$ bits of each agent's memory, in the general setting. Unlike the rooted setting, the agents are deployed arbitrarily. In **GeneralDisp**, we view the agents located at the same starting node as a single *group* and achieve rapid dispersion by having each group perform a HEO-DFS in parallel, sometimes merging groups. We show that by employing the group merge method given by Shintaku et al. [16], say *Zombie Method*, we can parallelize HEO-DFS by accepting an additive factor of $\log k$ to the space complexity and a multiplicative factor of $\log k$ to the time complexity. We have made substantial modifications to the Zombie Method to avoid conflicts between the function `Probe`() of HEO-DFS and the behavior of the Zombie Method.

As defined in Section 2, agents a with a.`settled` $= \top$ are called settlers, and the other agents are called explorers. In addition, in **GeneralDisp**, we classify explorers to two classes, *leaders* and *zombies*, depending on a variable `leader` $\in [0, \mathrm{id}_{\max}]$. We call an explorer a a leader if $a.\texttt{leader} = a.\texttt{ID}$, otherwise a zombie. Each agent a initially has $a.\texttt{leader} = a.\texttt{ID}$, so all agents are leaders at the start of an execution of **GeneralDisp**. As we will see later, a leader may become a zombie and a zombie will eventually become a settler, whereas a zombie never becomes a leader again, and a settler never becomes a leader or zombie again. Among the agents in $A(v,t)$, the set of leaders (resp., zombies, settlers) staying at v in time step t is denoted by $A_L(v,t)$ (resp., $A_Z(v,t), A_S(v,t)$). By definition, $A(v,t) = A_L(v,t) \cup A_Z(v,t) \cup A_S(v,t)$.

We introduce a variable $\texttt{level} \in \mathbb{N}$ to bound the execution time of **GeneralDisp**. We call the value of $a.\texttt{level}$ the *level* of agent a. The level of every agent is 1 initially. The pair $(a.\texttt{leader}, a.\texttt{level})$ serves as the group identifier: when agent a is a leader or settler, we say that a belongs to a group $(a.\texttt{leader}, a.\texttt{level})$. By definition, for any $(\ell, i) \in \mathbb{N}^2$, a group (ℓ, i) has at most one leader. A zombie does not belong to any group. However, when it accompanies a leader, it joins the HEO-DFS of that leader. We define a relationship $\prec$ between any two non-zombies a and b using these group identifiers as follows:

$$a \prec b \quad \Leftrightarrow \quad (a.\texttt{level} < b.\texttt{level}) \vee (a.\texttt{level} = b.\texttt{level} \wedge a.\texttt{leader} < b.\texttt{leader}).$$

We say that agent a is *weaker* than b if $a \prec b$, and that a is *stronger* than b otherwise.

Initially, all agents are leaders and each forms a group of size one. In the first time step, the strongest agent at each node turns all the other co-located agents into zombies (if exists). From then on, each leader performs a HEO-DFS while leading those zombies. For any leader a, we define the *territory* of a as

$$V_a = \{v \in V \mid \exists b \in A : \psi(v) = b \wedge b.\texttt{leader} = a.\texttt{ID} \wedge b.\texttt{level} = a.\texttt{level}\}.$$

Each time a leader a visits an unsettled node, it settles one of the accompanying zombies (if exists), giving it a's group identifier $(a.\texttt{leader}, a.\texttt{level})$. That is, a expands its territory. If a node outside a's territory is detected during the probing process of HEO-DFS, that node is considered unsettled even though it belongs to the territory of another leader. As a result, a may move forward to a node u that is inside another leader's territory. If that node u belongs to the territory of a weaker group, a incorporates the settler $\psi(u)$ into its own group by giving $\psi(u)$ its group identifier $(a.\texttt{leader}, a.\texttt{level})$. If a leader a encounters a stronger leader or a stronger settler during its HEO-DFS, a becomes a zombie and terminates its own HEO-DFS. If there is a leader at the current location $\nu(a)$ when a becomes a zombie, a joins the HEO-DFS of that leader. Otherwise, the agent a, now a zombie, chases a stronger leader by moving through the port $\psi(v).\texttt{next}$ at each node v. Unlike **RootedDisp**, a leader updates $\psi(v).\texttt{next}$ with the most recently used port even when it makes a backward move. This ensures that a catches up to a leader eventually, at which point a joins the HEO-DFS led by the leader.

Unlike **RootedDisp**, a leader does not settle itself at a node in the final stage of HEO-DFS. The leader a suspends the HEO-DFS if it visits an unsettled node but it has no accompanying zombies to settle at that time. A leader who has suspended the HEO-DFS due to the absence of accompanying zombies is called a *waiting leader*. Conversely, a leader with accompanying zombies is called an *active leader*. A waiting leader a resumes the HEO-DFS when a zombie catches up to a at $\nu(a)$. As we will see later, the execution of **GeneralDisp** ensures that all agents eventually become either waiting leaders or settlers, each residing at a distinct node. The agents have solved the dispersion once such a configuration is reached because thereafter no agent moves and no two agents are co-located.

When a leader a encounters a zombie z with the same level, a increments its level by one, and z resets its level to zero. This "level up" changes the identifier of a's group, i.e., from $(a.\texttt{ID}, i)$ to $(a.\texttt{ID}, i+1)$ for some i. By the definition of the territory, at this point, a loses

Table 2 Slot Assignments.

Slot Number	Role	Initiative	Pseudocode
Slot 1	Leader election	Leaders	2
Slot 2	Settle, increment level, etc.	Leaders	3
Slots 3	Move to join `Probe`()	Settlers	4
Slots 4–8	`Probe`()	Leaders	3
Slot 9–10	Chase for leaders	Zombies	5
Slot 11–12	Move forward/backward	Leaders	2

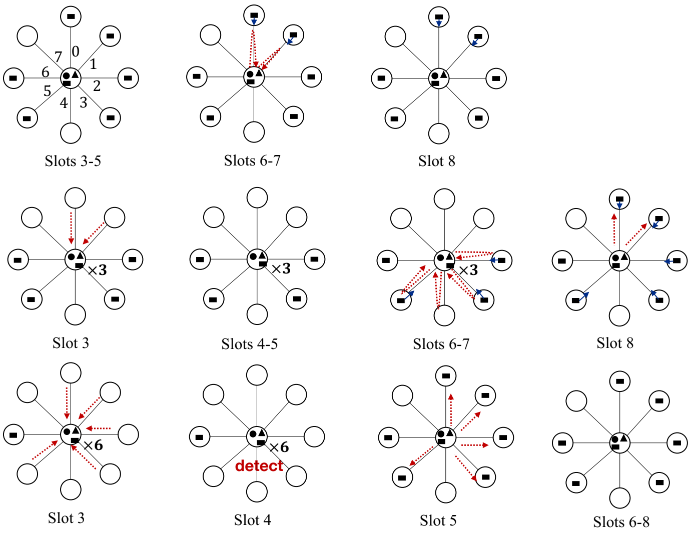
all nodes from its territory except for the current location. That is, each time a leader a increases its level, it restarts its HEO-DFS from the beginning. Note that this "level up" event also occurs when two leaders a, b ($b \prec a$) with the same level meet (and there is no stronger agent at the location) because then b becomes a zombie after it finds a stronger leader a, which results in the event that a leader a encounters a zombie with the same level, say b. We have the following lemma here.

▶ **Lemma 9.** *The level of an agent is always at most* $\log_2 k + 1$.

Proof. A level-up event requires one leader a and one zombie b with the same level. That zombie b will get level 0. Thereafter, b never triggers a level-up event again because the level of a leader is monotonically non-decreasing starting from level 1. Therefore, for any $i \geq 1$, the number of agents that can reach level i is at most $\lfloor k/2^{i-1} \rfloor$, leading the lemma. ◀

Therefore, each leader performs HEO-DFS at most $O(\log k)$ times. According to the analysis in Section 3, each HEO-DFS completes in $O(k \log \tau)$ time, which seems to imply that **GeneralDisp** finishes in $O((\log k) \cdot (k \log \tau)) = O(k \log^2 k)$ time. However, this analysis does not take into account the length of the period during which leaders suspend their HEO-DFS. Thus, it is not clear whether a naive implementation of the strategy described above would achieve the dispersion in $O(k \log^2 k)$ time. Following Shintaku et al. [16], we vary the speed of zombies chasing leaders based on a certain condition, which bounds the execution time by $O(k \log^2 k)$ time.

We give zombies different chasing speeds as follows. First, we classify zombies based on two variables $\texttt{level}_L$ and $\texttt{level}_S$ that each zombie manages. For any zombie z, we call $z.\texttt{level}_L$ and $z.\texttt{level}_S$ the *location level* and *swarm level* of z. When a leader z becomes a zombie, it initializes both $z.\texttt{level}_L$ and $z.\texttt{level}_S$ with its level, i.e., $z.\texttt{level}$. Thereafter, a zombie z copies the level of $\psi(\nu(z))$ to $z.\texttt{level}_L$ and updates $z.\texttt{level}_S$ to be $\max\{b.\texttt{level} \mid b \in A_Z(\nu(z))\}$ in every $O(1)$ time steps. Since a zombie only chases a leader with an equal or greater level, $z.\texttt{level}_S \leq z.\texttt{level}_L$ always holds. We say that a zombie z is *strong* if $z.\texttt{level}_S = z.\texttt{level}_L$; z is *weak* otherwise. Then, we exploit the assumption that the agents are synchronous and let weak zombies move twice as frequently as strong zombies to chase a leader. As we will prove later, this difference in chasing speed results in a desirable property of **GeneralDisp**, namely that $\min(\{a.\texttt{level} \mid a \in A_{AL}\} \cup \{z.\texttt{level}_L \mid z \in A_Z\})$ is monotone non-decreasing and increases by at least one in every $O(k \log \tau)$ steps, where A_{AL} is the set of active leaders and A_Z is the set of zombies both in the whole graph, until $A_{AL} \cup A_Z$ becomes empty. Thus, by Lemma 9, $A_{AL} \cup A_Z$ becomes empty and the dispersion is achieved in $O(k \log \tau \cdot \log k) = O(k \log^2 k)$ steps.



Slots 3-5 Slots 6-7 Slot 8

Slot 3 Slots 4-5 Slots 6-7 Slot 8

Slot 3 Slot 4 Slot 5 Slots 6-8

Figure 2 The behavior of explorers when their leader invokes `Probe()` at the center node w in **GeneralDisp**. A black circle, triangle, and rectangle represent a leader, a zombie, and a settler, respectively. The integers in the top left figure represents port numbers.

4.2 Implementation

In **GeneralDisp**, we group every 12 time steps into one unit, with each unit consisting of twelve *slots*. In other words, time steps $0, 1, 2, \ldots$ are classified into twelve slots. Specifically, each time step $t \geq 0$ is assigned to slot $(t \bmod 12) + 1$. For example, time step 26 is in slot 3, and time step 47 is in slot 12. Dividing all time steps into twelve slots helps to reduce the interference of multiple HEO-DFSs and allows us to set different "chasing speeds" for weak and strong zombies. Table 2 summarizes the roles of each slot.

Essentially, slots 1–2 are designated for leader election (i.e., group merging), slots 3–8 for probing, slots 9–10 for zombie chasing, and slots 11–12 for forward and backward movement in DFS traversal. It is important to note that settlers always stay at their home during slots 1–2 and 9–12. Hence, once a settler leaves its home, it returns within $O(1)$ steps. This is not the case in **RootedDisp**, where a settler in helping mode does not return home until its leader completes the probing process. In **GeneralDisp**, this frequent return home enables leaders to detect collisions with other groups: if a leader enters another group's territory, it will certainly notice the intrusion during the next slot 1, as it encounters a settler from that group.

Thus, the probing process in **GeneralDisp** slightly differs from that in **RootedDisp**. Consider a leader a_l starting the probing process at a node w (refer to Figure 2). The objective here is to identify any neighboring node of w that lies outside a_l's territory, if such exists. During slots 6–7, explorers at w visit its neighbors and return in parallel. If an explorer b encounters a settler s at a node $u \in N(w)$ in slot 6, b does not bring s back to w in slot 7. Instead, b requests s to enter helping mode, wherein s records the port number to w in the variable $s.\texttt{help} \in \mathbb{N} \cup \bot$ (A settler s is in helping mode if and only if $s.\texttt{mode} \neq \bot$). The helping settler s moves to w via port $s.\texttt{help}$ in the subsequent slot 3, joins the probing in slots 6-7, and returns to its home u again in slot 8. Leader a_l expects that the exact $\psi(w).\texttt{checked}$ helping settlers arrives at w in each slot 3. If this does not occur, a_l detects a non-territorial neighbor in slot 4 by checking the `pin` variable of the helping settlers at

w. Similar to **RootedDisp**'s probing process, the total number of explorers and helping settlers at w doubles until such a neighbor is detected, concluding the process in $O(\log \tau)$ steps. Subsequently, a_l reverts the helping settlers at w to non-helping mode by setting their `help` variable to $\perp$ and instructs them to return to their homes in slot 5.

As mentioned earlier, we differentiate the chasing speed of weak zombies and strong zombies. Specifically, weak zombies move in both slots 9 and 10, while strong zombies move only in slot 10.

We left the detailed implementation of **GeneralDisp** including pseudocodes and the complete proofs of its correctness and time complexity to the appendix due to space constraints. We give only a proof sketch here for the following main theorem.

▶ **Theorem 10.** *In the general setting, there exists an algorithm that solves the dispersion problem within $O(k \log \tau \cdot \log k)$ time using $O(\log(k + \Delta))$ bits of space per agent.*

▶ **Proof Sketch.** It suffices to show that $A_{AL} \cup A_Z$ becomes empty within $O(k \log \tau \cdot \log k)$ steps, at which point every agent is either a waiting leader or a settler, thereby achieving dispersion. We obtain this bound from Lemma 9 and the fact that $\alpha = \min(\{a.\texttt{level} \mid a \in A_{AL}\} \cup \{z.\texttt{level}_L \mid z \in A_Z\})$ increases by at least one in every $O(k \log \tau)$ time steps unless $A_{AL} \cup A_Z$ becomes empty (Lemma 15 in Appendix). We can prove Lemma 15 roughly as follows. Suppose $\alpha = i$. First, all weak zombies at location level i vanish within $O(k)$ steps as they move faster than leaders and strong zombies, eventually encountering a leader, higher-level settlers or stronger zombies. Thereafter, no new weak zombies at location level i are created. Subsequently, without weak zombies at location level i, waiting leaders at level i do not resume active HEO-DFS without increasing its level, which leads to the disappearance of active leaders at level i within $O(k \log \tau)$ steps. Finally, strong zombies chasing leaders at level i catch up to those leaders within $O(k)$ steps or find higher-level settlers, resulting in the increase of their location level. Hence, all zombies with location level i and active leaders at level i are eliminated within $O(k \log \tau)$ steps. ◀

5 For Further Improvement in Time Complexities

In the previous sections, we introduced nearly time-optimal dispersion algorithms: an $O(k \log \tau)$-time algorithm for the rooted setting and an $O(k \log^2 k)$-time algorithm for the general setting. This raises a crucial question: is it possible to develop a truly time-optimal algorithm, specifically an $O(k)$-time algorithm, even if it requires much more space? In this section, we affirmatively answer this question for the rooted setting. We present an $O(k)$-time algorithm that utilizes $O(\Delta + \log k)$ bits of space per agent. However, the feasibility of an $O(k)$-time algorithm in the general setting remains open.

We refer to the new algorithm as **RootedOpt** in this section. In **RootedDisp**, with $O(\log(k + \Delta))$ bits of space, each settler $\psi(w)$ cannot memorize the exact set of settled neighbors of w. Instead, it only remembers the maximum i such that the first i neighbors of w are settled. In contrast, **RootedOpt** allows each settler $\psi(w)$ to remember all settled neighbors of w using $O(\Delta)$ bits, which significantly helps to eliminate an $O(\log \tau)$ factor from the time complexity. However, somewhat surprisingly, both the design of the new algorithm and the analysis of its execution time are non-trivial.

Below, we outline the modifications made to **RootedDisp** to obtain **RootedOpt**. We expect that most readers will grasp the behavior of **RootedOpt** simply by reviewing the following key differences, while we provide the pseudocode for **RootedOpt** in the arXiv version [19].

- In **RootedOpt**, each settler $\psi(w)$ maintains an array variable $\psi(w)$.`checked` of size δ_w. Each element of $\psi(w)$.`checked`$[i]$ takes a value from the set $\{0, 1, \bot\}$. The assignment $\psi(w)$.`checked`$[i] = 0$ (respectively, 1) indicates that the neighbor $N(w, i)$ is unsettled (respectively, settled). The value $\bot$ is utilized exclusively during the probing process, meaning that the neighbor $N(w, i)$ has yet to be checked for its settled status. For any $c \in \{0, 1, \bot\}$, we define $U_c(u) = \{N(u, i) \mid i \in [0, \delta_w - 1], \psi(w).\texttt{checked}[i] = c\}$.
- Consider that the unique leader $a_{\max}$ invokes the probing process `Probe`() at a node w. (Remember that $a_{\max}$ is the unique leader that has the maximum identifier at the beginning of the execution.) In **RootedDisp**, the probing process terminates as soon as any agent finds an unsettled neighbor. However, in **RootedOpt**, the process only ends when all of w's neighbors are probed or when at least ℓ unsettled neighbors are found, where ℓ is the number of explorers. In the former case, the probing process is now complete: $U_\bot(w)$ is empty, and $U_0(w)$ equals the set of unsettled neighbors of w. In the latter case, the explorers go to distinct ℓ unsettled nodes and settle there, thereby achieving dispersion.
- Consider an agent a making a round trip $w \to u \to w$ during `Probe`(), where $u = N(w, p)$ for some $p \in [0, \delta_w - 1]$. If a does not encounter a settler at u, it simply sets $\psi(w)$.`checked`$[i]$ to 0. On the other hand, if a settler is found at u, a sets $\psi(w)$.`checked`$[i]$ to 1 and additionally sets $\psi(u)$.`checked`$[q]$ to 1, where $q \in [0, \delta_u - 1]$ is the port number such that $w = N(u, q)$. This modification ensures that $U_0(u)$ remains equal to the set of unsettled neighbors of u when the explorers go back to u.
- In **RootedDisp**, the leader $a_{\max}$ invokes `Probe`() after each forward or backward move. However, in **RootedOpt**, $a_{\max}$ only invokes `Probe`() after making a forward move. This change does not compromise the correctness of **RootedOpt** because when $a_{\max}$ makes a backward move to a node w, $\psi(w)$ accurately remembers its unsettled neighbors due to the modification mentioned earlier.

One might think that the probing process `Probe`() in **RootedOpt** could take longer time than in **RootedDisp**, as it only finishes after all neighbors of the current location have been probed or after finding ℓ unsettled neighbors, where ℓ is the number of explorers. Particularly, there seems to be a concern that during the probing at a node w, the number of agents, excluding $\psi(w)$, may not always double: this event occurs when some agents discover an unsettled neighbor. Despite that, we deny this conjecture at least asymptotically, that is, we have the following lemma.

▶ **Lemma 11.** *Assume that $a_{\max}$ invokes* `Probe`() *at node w during the execution of* **RootedOpt**, *and exactly ℓ explorers including $a_{\max}$ exists at the time. Then,* `Probe`() *finishes within $O(1) + \max(0, 2\lceil \log \tau - \log \ell \rceil)$ time.*

Proof. If $\ell \geq \tau$, we have $\ell \geq \min(\Delta, k) = \Delta$ because $\ell < k$. Then, the lemma trivially holds: `Probe`() finishes in a constant time. Thus, we consider the case $\ell < \tau$. Let t be the time step at which $a_{\max}$ invokes `Probe`() at a node w, and let $z = \lceil \log \tau - \log \ell \rceil + 1$. It suffices to show that $U_\bot(w, t') = \emptyset$ or $|U_0(w, t')| \geq \ell$ holds for some $t' \in [t, t + 2z + O(1)]$. Assume for contradiction that this does not hold. For any $r \in [0, z]$, we define $f(r) = X_r \cdot 2^{r-1} + Y_r$, where $X_r = |U_0(w, t + 2r)|$ and $Y_r = |A(w, t + 2r) \setminus \{\psi(w)\}|$. By definition, $f(0) = 0 \cdot 2^{0-1} + \ell = \ell$. Under the above assumption, for any $r = 0, 1, \dots, z - 1$, the agents in $A(w, t + 2r)$ move to distinct neighbors in $U_\bot(w, t + 2r)$ in time step $t + 2r$, and bring back all settlers they find, at most one for each neighbor, in time step $t + 2r + 1$. Let α be the number of those settlers i.e., $\alpha = Y_{r+1} - Y_r$. Note that $X_{r+1} - X_r = Y_r - \alpha$ holds here. Then, irrespective of α, we obtain

$$\begin{aligned}f(r+1) &= X_{r+1}\cdot 2^r + Y_{r+1} = (X_r + Y_r - \alpha)\cdot 2^r + Y_r + \alpha\\ &= X_r \cdot 2^r + (2^r - 1)(Y_r - \alpha) + 2\cdot Y_r \geq 2(X_r \cdot 2^{r-1} + Y_r) = 2f(r),\end{aligned}$$

where we use $2^r \geq 1$ and $Y_r - \alpha = X_{r+1} - X_r \geq 0$ in the above inequality. Therefore, we have $f(z) \geq \ell \cdot 2^z$, whereas we have assumed (for contradiction) that $X_z = |U_0(w, t+2z)| < \ell$, thus $f(z) = X_z \cdot 2^{z-1} + Y_z \leq (\ell - 1)2^{z-1} + Y_z$ holds. This yields $|A(w, t+2z) \setminus \{\psi(w)\}| = Y_z \geq \ell \cdot 2^z - (\ell-1)2^{z-1} = (\ell+1)2^{z-1} \geq (\ell+1)\tau/\ell > \tau$. Since $\tau = \min(\Delta, k)$, we have $\tau = \Delta$ or $\tau = k$. In the former case, $Y_k > \Delta$ agents at w are enough to visit all neighbors in $U_\perp(w, t+2z)$ in time step $t + 2z$, thus $U_\perp(w, t+2z+2) = \emptyset$ holds, a contradiction. In the latter case, there are $|A(w, t+2z)| \geq k+2$ agents in w at time step $t+2z$, a contradiction. Therefore, $U_\perp(w, t') = \emptyset$ or $|U_0(w, t')| \geq \ell$ holds at time step $t' \leq t + 2z + 2$. ◀

▶ **Theorem 12.** *In the rooted setting, algorithm* **RootedOpt** *solves the dispersion problem within $O(k)$ time using $O(\Delta + \log k)$ bits of space per agent.*

Proof. The unique leader $a_{\max}$ invokes `Probe`() only when it settles an agent, except for when $a_{\max}$ itself becomes settled. Therefore, $a_{\max}$ invokes `Probe` exactly $k-1$ times, with precisely $k-i$ explorers present at the i-th invocation. By Lemma 11, the total number of steps required for the $k-1$ executions of `Probe`() is at most $\sum_{\ell=1}^{k-1}(\log k - \log \ell + O(1)) = k \log k - (\log(k!) - \log k) + O(k) = O(k)$, where we apply Stirling's formula, i.e., $\log(k!) = k\log k - k + O(\log k)$. As demonstrated in Section 3, both forward and backward moves also require a total time of $O(k)$. Thus, **RootedOpt** completes in $O(k)$ time. Regarding space complexity, the array variable `checked` is the primary factor, needing $O(\Delta)$ bits per agent. Other variables require only $O(\log \Delta)$ bits. ◀

6 Discussion

It is worth mentioning that while HEO-DFS does not function in a *fully* asynchronous model, where movement between two nodes may require an unbounded period, it does not require a *fully* synchronous model in the rooted setting. Specifically, **RootedDisp** and **RootedOpt** can operate under an asynchronous scheduler if every movement of agents between nodes is *atomic*, i.e., each agent is always located at a node and never on an edge at any time step. Under this scheduler, after the probing process is completed at a node v, unsettled agents can wait for all helping settlers to leave v before they themselves depart. Since every movement is atomic, when unsettled agents visit the home node of one of these settlers, the settler has already returned. Thus, **RootedDisp** and **RootedOpt** functions under any fair scheduler that guarantees every movement is atomic. However, this move-atomicity is not sufficient for **GeneralDisp**, because this algorithm, designed for the general setting, differentiates the moving speeds of agents based on their roles – leader, strong zombie, or weak zombie – which inherently requires a fully synchronous scheduler.

References

1 Ankush Agarwalla, John Augustine, William K Moses Jr, Sankar K Madhav, and Arvind Krishna Sridhar. Deterministic dispersion of mobile robots in dynamic rings. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, pages 1–4, 2018. doi:10.1145/3154273.3154294.

2 John Augustine and William K. Moses Jr. Dispersion of mobile robots. *Proceedings of the 19th International Conference on Distributed Computing and Networking*, January 2018.

3 Prabhat Kumar Chand, Manish Kumar, Anisur Rahaman Molla, and Sumathi Sivasubramaniam. Fault-tolerant dispersion of mobile robots. In *Conference on Algorithms and Discrete Applied Mathematics*, pages 28–40, 2023. doi:10.1007/978-3-031-25211-2_3.

4 Archak Das, Kaustav Bose, and Buddhadeb Sau. Memory optimal dispersion by anonymous mobile robots. *Discrete Applied Mathematics*, 340:171–182, 2023. doi:10.1016/J.DAM.2023.07.005.

5 Shantanu Das. Graph explorations with mobile agents. *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*, pages 403–422, 2019. doi:10.1007/978-3-030-11072-7_16.

6 Ajay D Kshemkalyani and Faizan Ali. Efficient dispersion of mobile robots on graphs. In *Proceedings of the 20th International Conference on Distributed Computing and Networking*, pages 218–227, 2019. doi:10.1145/3288599.3288610.

7 Ajay D Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Fast dispersion of mobile robots on arbitrary graphs. In *International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics*, pages 23–40. Springer, 2019. doi:10.1007/978-3-030-34405-4_2.

8 Ajay D Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots in the global communication model. In *Proceedings of the 21st International Conference on Distributed Computing and Networking*, pages 1–10, 2020. doi:10.1145/3369740.3369775.

9 Ajay D Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots on grids. In *International Workshop on Algorithms and Computation*, pages 183–197. Springer, 2020. doi:10.1007/978-3-030-39881-1_16.

10 Ajay D Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots using global communication. *Journal of Parallel and Distributed Computing*, 161:100–117, 2022. doi:10.1016/J.JPDC.2021.11.007.

11 Ajay D. Kshemkalyani and Gokarna Sharma. Near-Optimal Dispersion on Arbitrary Anonymous Graphs. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, pages 8:1–8:19, 2021. doi:10.4230/LIPICS.OPODIS.2021.8.

12 Anisur Rahaman Molla, Kaushik Mondal, and William K Moses. Byzantine dispersion on graphs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 942–951. IEEE, 2021.

13 Anisur Rahaman Molla and William K Moses Jr. Dispersion of mobile robots: The power of randomness. In *International Conference on Theory and Applications of Models of Computation*, pages 481–500, 2019. doi:10.1007/978-3-030-14812-6_30.

14 Petrişor Panaite and Andrzej Pelc. Exploring unknown undirected graphs. *Journal of Algorithms*, 33(2):281–295, 1999. doi:10.1006/JAGM.1999.1043.

15 Vyatcheslav B Priezzhev, Deepak Dhar, Abhishek Dhar, and Supriya Krishnamurthy. Eulerian walkers as a model of self-organized criticality. *Physical Review Letters*, 77(25):5079, 1996.

16 Takahiro Shintaku, Yuichi Sudo, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Efficient dispersion of mobile agents without global knowledge. In *22nd International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 280–294, 2020. doi:10.1007/978-3-030-64348-5_22.

17 Yuichi Sudo, Daisuke Baba, Junya Nakamura, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. A single agent exploration in unknown undirected graphs with whiteboards. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 98(10):2117–2128, 2015. doi:10.1587/TRANSFUN.E98.A.2117.

18 Yuichi Sudo, Fukuhito Ooshita, and Sayaka Kamei. Brief announcement: Self-stabilizing graph exploration by a single agent. In *38th International Symposium on Distributed Computing*, 2024.

19 Yuichi Sudo, Masahiro Shibata, Junya Nakamura, Yonghwan Kim, and Toshimitsu Masuzawa. Near-linear time dispersion of mobile agents, 2024. arXiv:2310.04376, doi:10.48550/arXiv.2310.04376.

Algorithm 2 The behavior of a *leader* a.

```
1  while true do
2      /*************** Slot 1 begins ***************/
3      Let $w = \nu(a)$.
4      if $\exists b \in A_L(w) \cup A_S(w) : a \prec b$ then
5          $a.\mathtt{level}_L \leftarrow a.\mathtt{level}_S \leftarrow a.\mathtt{level}$
6          $a.\mathtt{leader} \leftarrow b.\mathtt{leader}$    // $a$ becomes a zombie and stops Algorithm 2
7      /*************** Slot 2 begins ***************/
8      if $A(w) \neq \{a\}$ then                 // $a$ is an active leader if $A(w) \neq \{a\}$
9          if $\psi(w) = \perp \ \vee \ \psi(w) \prec a$ then
10             if $\psi(w) = \perp$ then
11                 Settle one zombie in $A_Z(w)$ at $w$
12             $\psi(w).\mathtt{parent} \leftarrow a.\mathtt{parent}$          // Initially, $a.\mathtt{parent} = \perp$
13         if $\exists b \in A_Z(w) : a.\mathtt{level} = b.\mathtt{level}$ then
14             $(a.\mathtt{level}, b.\mathtt{level}) \leftarrow (a.\mathtt{level} + 1, 0)$
15             $\psi(w).\mathtt{parent} \leftarrow \perp$
16             $a.\mathtt{InitProbe} \leftarrow$ true
17         $(\psi(w).\mathtt{leader}, \psi(w).\mathtt{level}) \leftarrow (a.\mathtt{ID}, a.\mathtt{level})$
18         if $a.\mathtt{InitProbe} =$ true then          // Initially, $a.\mathtt{InitProbe} =$ true
19             $(\psi(w).\mathtt{next}, \psi(w).\mathtt{checked}, \psi(w).\mathtt{help}, \psi(w).\mathtt{done}) \leftarrow (\perp, -1, \perp, \mathbf{false})$
20             $a.\mathtt{InitProbe} \leftarrow$ false
21         $\mathtt{Probe}(a)$                                   // See Algorithm 3
22         if $\psi(w).\mathtt{done} =$ true then
23             /*************** Slot 11 begins ***************/
24             if $\psi(w).\mathtt{next} = \perp$ then
25                 $\psi(w).\mathtt{next} \leftarrow \psi(w).\mathtt{parent}$        // for backward move
26             All agents in $A(w) \setminus \{\psi(w)\}$ move to $N(w, \psi(w).\mathtt{next})$
27             /*************** Slot 12 begins ***************/
28             $a.\mathtt{parent} \leftarrow a.\mathtt{pin}$
29             $a.\mathtt{InitProbe} \leftarrow$ true
```

A Detail Implementation of General Dispersion

The pseudocode for the **GeneralDisp** algorithm is shown in Algorithms 2, 3, 4, and 5. In slots 1, 2, 4–8, 11, and 12, agents operate only under the instruction of a leader. Algorithms 2 and 3 define how each leader a operates and gives instructions in those slots. Algorithm 4 defines the behavior of settlers in slot 3. Algorithm 5 specifies the behavior of zombies in slots 9 and 10. Note that each agent needs to manage an $O(1)$-bit variable to identify the slot of the current time step, but for simplicity, the process related to its update is not included in the pseudocode because it can be implemented in a naive way.

First, we explain the behavior of a leader a. Let w be the node where a is located in slot 1. We make leader election in slot 1 (lines 4–6). Leader a becomes a zombie when it finds a stronger leader or settler on w. If a becomes a zombie, it no longer runs Algorithms 2 and 3, and runs only Algorithm 5. Consider that a survives the leader election in slot 1. In slot 2, if there are no agents other than a on w, a is a waiting leader and does nothing until the next slot 1. Otherwise, the leader a (i) settles one of the accompanying zombies if w is unsettled,

Algorithm 3 `Probe`(a).

```
/*************** Slot 4 begins ***************/
Let $w = \nu(a)$.
$\psi(w).\mathtt{next}(w) \leftarrow \begin{cases} \min P & \text{if } P \neq \emptyset \\ \bot & \text{otherwise,} \end{cases}$
   where $P = [0, \psi(w).\mathtt{checked}] \setminus \{b.\mathtt{pin} \mid b \in A_S(w) \setminus \{\psi(a)\}\}$
$b.\mathtt{help} \leftarrow \bot$ for all $b \in A_S(w)$ with $b \prec a$.
Let all agents $b \in A_S(w)$ with $b \prec a$ go back to their homes
if $\psi(w).\mathtt{next} \neq \bot \vee \psi(w).\mathtt{checked} = \delta_w - 1$ then
    /*************** Slot 5 begins ***************/
    Execute $b.\mathtt{help} \leftarrow \bot$ for each $b \in A_S(w) \setminus \{\psi(w)\}$
    Let all agents in $A_S(w) \setminus \{\psi(w)\}$ go back to their homes.
    $\psi(w).\mathtt{done} \leftarrow$ true
else
    /*************** Slot 6 begins ***************/
    Let $\{a_1, a_2, \dots, a_x\}$ be the set of agents in $A(w) \setminus \{\psi(w)\}$
    Let $\Delta' = \min(x, \delta_w - 1 - \psi(w).\mathtt{checked})$
    Let $u_i = N(w, i + \psi(w).\mathtt{checked})$ for $i = 1, 2, \dots, \Delta'$
    for each $a_i \in \{a_1, a_2, \dots, a_{\Delta'}\}$ in parallel do
        $a_i$ moves to $u_i$.
        /*************** Slot 7 begins ***************/
        if $(a_i.\mathtt{leader}, a_i.\mathtt{level}) = (\psi(u_i).\mathtt{leader}, \psi(u_i).\mathtt{level})$ then
            $a_i.\mathtt{found} \leftarrow$ true
            $\psi(u_i).\mathtt{help} \leftarrow a_i.\mathtt{pin}$
        else
            $a_i.\mathtt{found} \leftarrow$ false
        Move to $N(u_i, a_i.\mathtt{pin})$
    /*************** Slot 8 begins ***************/
    if $\exists i \in [1, \Delta'] : a_i.\mathtt{found} =$ false then
        $\psi(w).\mathtt{next} \leftarrow i + \psi(w).\mathtt{checked}$
    $\psi(w).\mathtt{checked} \leftarrow \psi(w).\mathtt{checked} + \Delta'$
    Let all agents in $A_S(w) \setminus \{\psi(w)\}$ go back to their homes.
```

Algorithm 4 The behavior of a *settler* s in Slot 3.

```
/*************** Slot 3 begins ***************/
Move to $N(\nu(s), s.\mathtt{help})$ if $s.\mathtt{help} \neq \bot$.
```

Algorithm 5 The behavior of a *zombie* z in Slots 9 and 10.

```
62 /*************** Slot 9 begins ***************/
63 $(z.\texttt{level}_L, z.\texttt{level}_S) \leftarrow (\psi(w).\texttt{level}, \max\{z'.\texttt{level} \mid z' \in A_Z(\nu(z)\})$
64 if $A_L(\nu(z)) = \emptyset$ and $z$ is a weak zombie then
65     Move to $N(\nu(z), \psi(\nu(z)).\texttt{next})$
66 /*************** Slot 10 begins ***************/
67 if $A_L(\nu(z)) = \emptyset$ then
68     Move to $N(\nu(z), \psi(\nu(z)).\texttt{next})$
```

(ii) updates its level if it finds a zombie with the same level, and (iii) gives the settler $\psi(w)$ its group identifier $(a.\texttt{ID}, a.\texttt{level})$ (lines 9–17). Note that settlers may leave their homes only in slots 3–8 (to join `Probe`()), thus a can correctly determine whether $\psi(w) = \bot$ or not here (lines 9–10). If $\psi(w) \prec a$, this procedure incorporates $\psi(w)$ into a's group, i.e., expands the territory of a. Each leader a manages a flag variable $a.\texttt{InitProbe} \in \{\textbf{false}, \textbf{true}\}$, initially set to **true**. This flag is raised each time a requires probing, i.e., after it makes a forward or backward move (line 29), and when it increases its level (line 16). If the flag is raised, it initializes the variables used for `Probe`(), say $\psi(w).\texttt{next}$, $\psi(w).\texttt{checked}$, $\psi(w).\texttt{help}$, and $\psi(w).\texttt{done}$ in slot 2 (line 19).

Thereafter, a invokes `Probe`() at the end of slot 2. This subroutine runs in slots 4–8. While `Probe`() in **RootedDisp** returns the control to the main function after completing the probing, i.e., determining whether or not an unsettled neighbor exists, `Probe`() in **GeneralDisp** returns the control each time slot 8 ends even if it does not complete the probing. Consider that there are $x - 1$ accompanying zombies when a leader a begins the probing. First, a leader a and the $x - 1$ accompanying zombies join the probing. Each of them, say b, moves from a node w to one of its neighbors $u \in N(w)$ in slot 6 (line 47) and goes back to w in slot 7 (line 54). If b finds a settler in the same group at u, it sets $\psi(u).\texttt{help}$ to $b.\texttt{pin}$ (line 51). As long as $s.\texttt{help} \neq \bot$, a settler s at a node v goes to a neighbor $N(v, s.\texttt{help})$ in slot 3 (line 60, Algorithm 4). Hence, in the next slot 3, that settler $\psi(u)$ goes to w. If there are $2x$ agents at w excluding $\psi(w)$, those $2x$ agents perform the same process in the next slots 6 and 7, that is, they go to unprobed neighbors, update the `help` of settlers in the same group (if exists), and go back to w. In slot 8, a sends the helping settlers back to their home. The number of agents joining the probing at w, i.e., $|A(w) \setminus \{\psi(w)\}|$, doubles at each iteration of this process until they find a node without a settler in the same group or finish probing all neighbors in $N(w)$. Thus, like **RootedDisp**, the probing finishes in $O(\log \tau)$ time steps. At this time, $\psi(w).\texttt{next} = \bot$ holds if all neighbors in $N(w)$ are settled by settlers in the same group. Otherwise, $N(w, \psi(w).\texttt{next})$ is unsettled or settled by a settler in another group. Then, in the next slot 5, a resets the `help` of all settlers at w to $\bot$ except for $\psi(w)$, lets them go back to their homes, and sets $\psi(w).\texttt{done}$ to **true**, indicating that the probing is done (lines 38–40). The probing process described above may be prevented by a stronger leader b when b visits a node $v \in N(w)$ such that $\psi(v)$ belongs to a's group and $\psi(v).\texttt{help} \neq \bot$. Then, b incorporates $\psi(v)$ into b's group, and set $\psi(v).\texttt{help}$ to $\bot$ (line 19), so $\psi(v)$ never goes to w to help a's probing. However, this event actually speeds up a's probing: a identifies this event when noticing that $\psi(v)$ does not arrive at w in the next slot 4. As a result, a can set $\psi(w).\texttt{next}$ to p where $N(w, p) = v$ (lines 32–34).

Note that, even during the probing process at node w, leader a might become a zombie if it meets a stronger leader b in slot 1. Some settlers might then move to w in the next slot 3 to help a, not knowing a is now a zombie. In these situations, b changes the `help` of these settlers to $\perp$ and sends them back to their homes in slot 4. Thereafter, those settlers remain at their home at least until they are incorporated into another group.

If a leader a at w observes $\psi(w).\mathtt{done} = \mathbf{true}$, it makes a forward or backward move in slot 11 (lines 22–29). Each time a makes a forward or backward move to a node u, it remembers $a.\mathtt{pin}$ in $a.\mathtt{parent}$ after the move (line 28). This port number will be stored on the variable $\psi(u).\mathtt{parent}$ when a settles a zombie on u or a incorporates $\psi(u)$ from the territory of another group. Note that this event occurs only when the last move is forward. Thus, like **RootedDisp**, a constructs a DFS tree in its territory. It is inevitable to use a variable $a.\mathtt{parent}$ tentatively since $a.\mathtt{pin}$ is updated every step by definition of a special variable $\mathtt{pin}$ and a may become a waiting leader after moving to u. Unlike **RootedDisp**, a records the most recently used port to move in $\psi(\nu(a)).\mathtt{next}$ even when it makes a backward move (lines 24–25). This allows a zombie to chase a leader.

The behavior of zombies in slots 9 and 10 is very simple (lines 61–67). A zombie always updates its location and swarm levels in slot 9 (line 62). A zombie z not accompanying a leader always chases a leader by moving through the port $\psi(\nu(z)).\mathtt{next}$. As mentioned earlier, we differentiate the chasing speed of weak zombies and strong zombies. Specifically, weak zombies move in both slots 9 and 10, while strong zombies move only in slot 10 (lines 63–67).

▶ **Lemma 13.** *The location level of a zombie is monotonically non-decreasing.*

Proof. Neither a leader nor a settler decreases its level in **GeneralDisp**. When a zombie z does not accompany a leader, it chases a leader through port $\psi(\nu(z)).\mathtt{next}$. This port $\psi(\nu(z)).\mathtt{next}$ is updated only if a leader makes a forward or backward move from $\nu(z)$, and the leader updates the level of $\psi(N(\nu(z), \psi(\nu(z)).\mathtt{next}))$ if it is smaller than its level. Thus, a zombie never decreases its location level by chasing a leader. When a zombie z accompanies a leader, the leader copies its level to $\psi(\nu(z)).\mathtt{level}$ in slot 2, which is copied to $z.\mathtt{level}_L$ in slot 8. The leader that z accompanies may change but does not change to a weaker leader. Thus, a zombie never decreases its location level when accompanying a leader. ◀

B Proofs of Theorem 10

Remember that A_Z and A_{AL} are the set of zombies and the set of active leaders, respectively, in the whole graph. We have the following lemma.

▶ **Lemma 14.** *For any $i \geq 0$, the number of weak zombies with a location level i is monotonically non-increasing starting from any configuration where* $\min(\{a.\mathtt{level} \mid a \in A_{AL}\} \cup \{z.\mathtt{level}_L \mid z \in A_Z\}) = i$.

Proof. Let C be a configuration where $\min(\{a.\mathtt{level} \mid a \in A_{AL}\} \cup \{z.\mathtt{level}_L \mid z \in A_Z\}) = i$. When a leader with level i becomes a zombie, its location level is i (line 5). So, a leader with level i may become a strong zombie with a location level i but never becomes a weak zombie with a location level i. The swarm level of a zombie decreases only when the zombie accompanies a leader (and this leader settles another zombie). Thus, a strong zombie with a location level i that does not accompany a leader cannot become a weak zombie without increasing its location level. Moreover, starting from C, a strong zombie with location level i must increase its location level when it encounters a leader in slot 1. Hence, the number of weak zombies with a location level i is monotonically decreasing. ◀

▶ **Lemma 15.** $\min(\{a.\mathtt{level} \mid a \in A_{AL}\} \cup \{z.\mathtt{level}_L \mid z \in A_Z\})$ *is monotone non-decreasing and increases by at least one in every $O(k \log \tau)$ time steps unless $A_{AL} \cup A_Z$ becomes empty.*

Proof. Let i be an integer $i \geq 0$ and C a configuration where $\min(\{a.\mathtt{level} \mid a \in A_{AL}\} \cup \{z.\mathtt{level}_L \mid z \in A_Z\}) = i$. It suffices to show that leaders with level i and zombies with location level i disappear in $O(k \log \tau)$ time steps starting from C.

Consider an execution starting from C. By Lemma 14, a weak zombie with location level i is never newly created in this execution. Let z be any weak zombie with a location level i that does not accompany a leader in a configuration C. In every 12 slots, z moves twice, while a strong zombie and a leader move only once, excluding the movement for the probing. Therefore, z catches up to a strong zombie and becomes strong too, catches up to a leader with level i, or increases its location level in $O(k)$ time steps. When z catches up to a leader, it joins the HEO-DFS of the leader, or this leader becomes a zombie. In the latter case, z becomes a strong zombie. Thus, z settles or becomes a strong zombie (with the current leader) in $O(k \log \tau)$ time steps. Therefore, the number of weak zombies with location level i becomes zero in $O(k \log \tau)$ steps. After that, no waiting leader with level i resumes its HEO-DFS without increasing its level because there is no weak zombie with location level i. Therefore, every active leader with location level i becomes a zombie with location level at least $i + 1$ or a waiting leader in $O(k \log \tau)$ steps. Thus, active leaders with location level i also disappear in $O(k \log \tau)$ steps. From this time, no leader moves in the territory of a group with level i or less. Hence, every strong zombie with location level i increases its location level or catches up to a waiting leader. Since the level of a waiting leader is at least i, the latter event also increases z's level by at least one. ◀

Since an agent in **GeneralDisp** manages only a constant number of variables, each with $O(\log(k + \Delta))$ bits, Lemmas 9 and 15 yield Theorem 10.

The Power of Abstract MAC Layer: A Fault-Tolerance Perspective

Qinzi Zhang
Boston University, MA, USA

Lewis Tseng
UMass-Lowell, MA, USA

Abstract

This paper studies the power of the "*abstract MAC layer*" model in a single-hop asynchronous network. The model captures primitive properties of modern wireless MAC protocols. In this model, Newport [PODC '14] proves that it is impossible to achieve deterministic consensus when nodes may crash. Subsequently, Newport and Robinson [DISC '18] present randomized consensus algorithms that terminate with $O(n^3 \log n)$ expected broadcasts in a system of n nodes. We are not aware of any results on other fault-tolerant distributed tasks in this model.

We first study the computability aspect of the abstract MAC layer. We present a *wait-free* algorithm that implements an atomic register. Furthermore, we show that in general, k-set consensus is impossible. Second, we aim to minimize storage complexity. Existing algorithms require $\Omega(n \log n)$ bits. We propose two *wait-free* approximate consensus and two *wait-free* randomized binary consensus algorithms that only need *constant* storage complexity (except for the phase index). One randomized algorithm terminates with $O(n \log n)$ expected broadcasts. All our algorithms are *anonymous*, meaning that at the algorithm level, nodes do not need to have a unique identifier.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Abstract MAC Layer, Computation Power, Consensus

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.39

Related Version *Full Version*: `https://arxiv.org/abs/2408.10779` [42]

Funding *Lewis Tseng*: This material is based upon work partially supported by the National Science Foundation under Grant CNS-2334021.

Acknowledgements The authors want to thank the anonymous reviewers for insightful comments. The work was partially done when Lewis Tseng was affiliated with Boston College and Clark University.

1 Introduction

This paper studies fault-tolerant primitives, with the focus on the aspect of wireless links in a single-hop asynchronous network. We adopt the "*abstract MAC layer*" model [33, 34, 25], which captures the basic properties guaranteed by existing wireless MAC (medium access control) layers such as TDMA (time-division multiple access) or CSMA (carrier-sense multiple access). Even though the abstraction does not model after any specific existing MAC protocol, the abstract MAC layer still serves an important goal – the separation of high-level algorithm design and low-level logic of handling the wireless medium and managing participating nodes. This separation helps identify principles that fills the gap between theory and practice in designing algorithms that can be readily deployed onto existing MAC protocols [33, 34]. In fact, recent works in the networking community propose approaches to implement the abstract MAC layer in more realistic network conditions, e.g., dynamic systems [41], dynamic SINR channels [40], and Rayleigh-Fading channels [39].

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 39; pp. 39:1–39:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Consider an asynchronous network [10, 31] in which messages may suffer an arbitrary delay. Compared to conventional message-passing models [10, 31], the abstract MAC layer has two key characteristics (formal definition in Section 2):

- Nodes use a broadcast primitive which sends a message to all nodes that have *not* crashed yet, and triggers an acknowledgement upon the completion of the broadcast.
- Nodes do *not* have a priori information on other participating nodes.

The second characteristic is inspired by the observation that in a practical large-scale deployment, it is difficult to configure and manage all the connected devices so that they have the necessary information about other nodes. This assumption makes it difficult to port algorithms from conventional message-passing models to the abstract MAC layer, as these algorithms typically require the knowledge of other nodes and/or the size of the system. In fact, Newport and Robinson prove [33] that in message-passing models it is impossible to solve deterministic and randomized consensus, even if there is no fault, and nodes are assumed to have a constant-factor approximation of the network size.

In the abstract MAC layer model, Newport [34] proves that deterministic consensus is impossible when nodes may crash. Subsequently, Newport and Robinson [33] propose randomized consensus algorithms. We are not aware of any study on other fault-tolerant primitives. This paper answers the following two fundamental problems:

- Can we implement other fault-tolerant primitives?
- How do we minimize storage complexity when designing fault-tolerant primitives?

First Contribution. In Herlihy's wait-free hierarchy [23], the consensus number defines the "power" of a shared object (or primitive). An object has a consensus number c, if it is possible for $\leq c$ nodes to achieve consensus using the object and atomic registers, *and* it is not possible for $c+1$ nodes to do so. For example, an atomic register has consensus number 1, whereas consensus and compare-and-swap have consensus number ∞. The proof in [34] implies that any objects with consensus number ≥ 2 cannot be implemented in the abstract MAC layer. The natural next step is to understand whether objects with consensus number 1 can be implemented in the abstract MAC layer.

We first show that the abstract MAC layer is fundamentally related to the *store-collect object* [7, 8] by presenting a simple *wait-free* algorithm to implement the object in the abstract MAC layer. "Stacking" the constructions in [8] on top of our store-collect object solves many well-known computation tasks, e.g., registers, counters, atomic snapshot objects, and approximate, and randomized consensus. That is, we provide a wait-free approach to implement *some primitives with consensus number 1 in the abstract MAC layer.*

Next, we identify that *not all* primitives with consensus number 1 can be implemented. In particular, we prove that in a system of n nodes, $(n-1)$-set consensus is impossible to achieve in the abstract MAC layer model. This implies that other similar objects, like write-and-read-next objects [16], cannot be implemented as well.

Second Contribution. From a more practical perspective, we study *anonymous* and *storage-efficient* fault-tolerant primitives. First, *anonymous* algorithms do not assume unique node identity, and thus lower efforts in device configuration and deployment. Second, most wireless devices are made small; hence, naturally, they are not equipped with abundant storage capacity, and storage-efficiency is an important factor in practical deployment.

Table 1 compares the state-of-the-art algorithm NR18 [33] and our algorithms. All the randomized consensus algorithms work for binary inputs and all algorithms are *wait-free*. The time complexity is measured as the expected number of broadcasts needed for all fault-free nodes to output a value. For our algorithms, "values" can be implemented using an integer or a float data type in practice. The exact size of the values will become clear later.

Table 1 Consensus in abstract MAC layer.
• The bottom four rows present our algorithms, whereas NR18 is the algorithm from DISC '18 [33].
• For approximate consensus, the convergence rate identifies the ratio that the range of fault-free nodes' states decreases after each asynchronous round. The smaller the ratio, the faster the convergence.

	consensus	storage complexity	note
NR18 [33]	randomized	$\Theta(n \log n)$ bits	$O(n^3 \log n)$ expected broadcasts
MAC-RBC	randomized	8 values, 4 Booleans	$O(2^n)$ expected broadcasts
MAC-RBC2	randomized	12 values, 5 Booleans	$O(n \log n)$ expected broadcasts
MAC-AC	approximate	4 values, 1 Boolean	convergence rate 1/2
MAC-AC2	approximate	2 values, 1 Boolean	convergence rate $1 - 2^{-n}$

Due to space limits, we focus only on our randomized algorithms and present our approximate agreement algorithms along with the full analysis in the technical report [42].

2 Preliminary

Related Work. We first discuss prior works in the abstract MAC layer model. The model is proposed by Kuhn, Lynch and Newport [25]. They present algorithms for multi-message broadcast, in which multiple messages may be sent at different times and locations in a multi-hop network communicating using the abstract MAC layer. Subsequent works [32, 24, 20] focus on *non-fault-tolerant* tasks, including leader election and MIS.

The closest works are by Newport [34] and Newport and Robinson [33]. Newport presents several impossibilities for achieving *deterministic* consensus when nodes may crash [34]. Newport and Robinson [33] present a randomized consensus algorithm that terminates after $O(n^3 \log n)$ broadcasts w.h.p. In their algorithm, nodes need to count the number of acknowledgements received from unique nodes and determine when to safely output a value. As a result, their algorithm requires storage space $\Theta(n \log n)$ bits and the knowledge of identities to keep track of unique messages. An accompanied (randomized) approach of assigning node identities with high probability is also proposed in [33]. Tseng and Sardina [36] present Byzantine consensus algorithms in the abstract MAC layer model, but they assume the knowledge of an upper bound on n and unique identities. Our consensus algorithms do not rely on identities; hence, fundamentally use different techniques.

Fault-tolerant consensus has been studied in various models that assume message-passing communication links [10, 31]. We consider a different communication model; hence, the techniques are quite different. An important distinction is that with asynchronous message-passing, it is impossible to implement a *wait-free* algorithm [23]. Furthermore, nodes require accurate information on the network size [33].

Di Luna et al. have a series of works on anonymous dynamic network [27, 28, 30, 29, 18, 17]. They do not assume any failures. A series of papers [21, 2, 13] study a related problem, called consensus with unknown participants (CUPs), where nodes are only allowed to communicate with other nodes whose identities have been provided by some external mechanism. Our consensus algorithms do not need unique node identities. Failure detectors are used in [1, 35, 12] to solve consensus with anonymous nodes. We do not assume a failure detector.

Model. We consider a static asynchronous system consisting of n nodes, i.e., we do not consider node churn. Each node is assumed to have a unique identifier; hence, the set of nodes is also denoted as the set of their identifiers, i.e., $\{1, \ldots, n\}$. For brevity, we often denote it by $[n]$. Our construction of store-collect requires identifiers due to its semantics. Our approximate and randomized algorithms are anonymous, and do *not* assume node identifiers. The identifiers are used only for analysis.

We consider the crash fault model in which any number of nodes may fail. A faulty node may crash and stop execution at any point of time. The adversary may control faulty behaviors and the message delays. Nodes that are not faulty are called *fault-free* nodes.

In a single-hop network with abstract MAC layer [25, 34, 33], nodes communicate using the `mac-broadcast` primitive, which eventually delivers the message to all the nodes that have not crashed yet, including i itself. Moreover, at some point after the `mac-broadcast` has succeeded in delivering the message, the broadcaster receives an acknowledgement, representing that the `mac-broadcast` is complete. The broadcaster *cannot* infer any other information from the acknowledgment, not the system size n, nor the identities of other nodes. A crash may occur during the `mac-broadcast`, which leads to inconsistency. That is, if a broadcaster crashes, then some nodes might receive the message while others do not.

The key difference between message-passing and abstract MAC layer models is that in the message-passing model, sender requires explicit responses, which is the main reason that this model does not support wait-free algorithms and requires a priori information on participating nodes. In other words, abstract MAC layer allows us to design primitives with stronger properties due to the implication from the acknowledgement. This paper is the first to identify how to use its "power" to implement (some) primitives with consensus number 1.

For our store-collect and approximate consensus algorithms, the adversary may enforce an arbitrary schedule for message delivery and crashes. For our randomized algorithms, we assume the *message oblivious (or value oblivious)* adversary as in [33]. That is, the adversary does *not* know the private states of each node (process states or message content).

Algorithm Presentation and Message Processing. Each node is assumed to take steps sequentially (a single-thread process). Each line of the pseudo-code is executed *atomically*, except when calling `mac-broadcast`, since this primitive is handled by the underlying abstract MAC layer. Each algorithm also has a message handler that processes incoming messages. Our algorithms assume that (i) the message handler is triggered whenever the underlying layer receives a message and sends an interrupt; (ii) there is only one message handler thread, which processes messages one by one, i.e., the underlying layer has a queue of pending messages; and (iii) the handler has a priority over the execution of the main thread. The third assumption implies the following observation, which is important for ensuring the correctness of our algorithms:

▶ Remark 1. At the point of time when the main thread starts executing a line of the pseudo-code, there is *no* pending message to be processed by the handler.[1]

It is possible that *during* the execution of a line in the main thread, the underlying layer sends an interrupt. The message handler will process these messages *after* the completion of that particular line of code due to the assumption of atomic execution. The only exception is the call to `mac-broadcast`. Messages can still be received and processed when a broadcaster is waiting for the acknowledgement from the abstract MAC layer.

[1] This assumption is not needed in prior works [34, 33], because their algorithm design is fundamentally different from ours. On a high-level, their algorithms proceed in an atomic block, whereas our algorithms have shared variables between the main thread and multiple message handlers. The assumption captures the subtle interaction between them.

3 Abstract MAC Layer: Computability

From the perspective of computability, asynchronous point-to-point message-passing model is fundamentally related to linearizable shared objects [22]. However, it was pointed out that register simulation in conventional point-to-point models like ABD [6] is "thwarted" [33]. In other words, this observation indicated that the computability of the abstract MAC layer remained an *open* problem. We fill the gap by presenting a framework of implementing some linearizable shared objects with consensus number 1.

Our Insight. In the point-to-point model, ABD requires the communication among a quorum, because "information kept by a quorum" ensures that the information is *durable* and *timely* in quorum-based fault-tolerant designs. Durable information means that the information is not lost, even after node crashes. Timely information means that the information satisfies the real-time constraint, i.e., after the communication with a quorum is completed, others can learn the information by contacting any quorum of nodes.

Our important observation is that the `mac-broadcast` achieves *both* goals upon learning the acknowledgement. That is, after the broadcaster learns that the broadcast is completed, it can infer that the message is both durable and timely.

Durability and timeliness are indeed sufficient for ensuring "regularity," which can then be used to implement linearizable shared objects (as will be seen in Theorem 3). We next present a construction of store-collect objects.

Store-Collect Object. A store-collect object [7, 8] provides two operations (or interfaces) at node i: (i) $\textsc{Store}_i(v)$: store value v into the object; and (ii) $\textsc{Collect}_i()$: collect the set of "most recent" values (of the object) from each node. The returned value is a *view* V – a set of (v_j, j) tuples where j is a node identity and v_j is its most recent stored value. For each j, there is at most one tuple of $(*, j)$ in V. With a slight abuse of notation, $V(j) = v_j$ if $(v_j, j) \in V$; otherwise, $V(j) = \perp$.

To formally define store-collect, we first discuss a useful notion. A *history* is an execution of the store-collect object, which can be represented using a partially ordered set $(H, <_H)$. Here, H is the set of invocation (inv) and response ($resp$) events of the $\textsc{Store}$ and $\textsc{Collect}$ operations, and $<_H$ is an irreflexive transitive relation that captures the real-time "occur-before" relation of events in H. Formally, for any two events e and f, we say $e <_H f$ if e occurs before f in the execution. For two operations op_1 and op_2, we say that op_1 precedes op_2 if $resp(op_1) <_H inv(op_2)$.

Every value in $\textsc{Store}$ is assumed to be unique (this can be achieved using sequence numbers and node identifiers). A node can have at most one pending operation. Given views V_1 and V_2 returned by two $\textsc{Collect}$ operations, we denote $V_1 \preceq V_2$, if for every $(v_1, j) \in V_1$, there exists a v_2 such that (i) $(v_2, j) \in V_2$; and (ii) either $v_1 = v_2$ or the invocation of $\textsc{Store}_j(v_2)$ occurs after the response of $\textsc{Store}_j(v_1)$. That is, from the perspective of node j, v_2 is more recent than v_1. We then say that a history σ satisfies *regularity* if:

- For each $\textsc{Collect}()$ $c \in \sigma$ that returns V and for each node j, (i) if $V(j) = \perp$, then no $\textsc{Store}$ by j precedes c in σ; and (ii) if $V(j) = v$, then $\textsc{Store}_j(v)$'s invocation precedes c's response, and there does not exist $\textsc{Store}_j(v')$ such that $v' \neq v$, and $\textsc{Store}_j(v')$'s response occurs after $\textsc{Store}_j(v)$'s response and before c's invocation.
- Consider any pair of two $\textsc{Collect}$'s in history σ, c_1 and c_2, which return views V_1 and V_2, respectively. If c_1 precedes c_2, then $V_1 \preceq V_2$.

▶ **Definition 2** (Store-Collect). *An algorithm correctly implements the store-collect object if every execution of the algorithm results into a history that satisfies regularity.*

Our Wait-free Construction of Store-Collect. To achieve regularity, each stored value has to be durable and timely. If a value is not durable, then the first condition for regularity may be violated. If a value is not timely, then the second condition may be violated. Moreover, any current information needs to be known by subsequent COLLECT's, potentially at other nodes. These observations together with the aforementioned insight of the `mac-broadcast` primitive give us a surprisingly straightforward construction. Our algorithm MAC-SC is presented in Algorithm 1.

Each node i keeps a local variable $view_i$, which is a set of values – one value for each node (that is known to node i so far). With a slight abuse of terminology, we use $C = A \cup B$ to denote the merge operation of two views A and B, which returns a view C that contains the newer value from each node. Since each node can have at most one pending operation and each value is unique, the notion of "newer" is well-defined. For brevity, the sequence number is omitted in the notation.

For STORE$_i(v)$, node i first adds the value v to form a new view, and uses `mac-broadcast` to inform others about the new view. Because this information is both durable and timely upon the completion of the broadcast, regularity is satisfied. Since the broadcast delivers the message to the broadcaster as well, (v, i) is added to $view_i$ at line 8. For COLLECT$_i()$, it is similar except that the broadcast view is the current local view at node i. Upon receiving a new view (from the incoming message with the STORE tag), i simply merges the new view and its local view $view_i$.

Algorithm 1 MAC-SC: Steps at each node i.

```
Local Variable: /* It can be accessed by any thread at $i$. */
    $view_i$                                              ▷view, initialized to $\emptyset$
When $Store_i(v)$ is invoked:
 1: $currentView_i \leftarrow view_i \cup \{(v, i)\}$
 2: mac-broadcast(STORE, $currentView_i$)
 3: return ACK              ▷STORE is completed
When $Collect_i()$ is invoked:
 4: $currentView \leftarrow view_i$
 5: mac-broadcast(STORE, $currentView$)
 6: return $currentView$
// Background message handler
 7: Upon receive(STORE, $view$) do
 8:     $view_i \leftarrow view_i \cup view$
```

▶ **Theorem 3.** *MAC-SC implements the store-collect object.*

Proof Sketch.

Property I. Consider a COLLECT$_i()$ operation c that returns V. For each node j, consider two cases:

- $V(j) = \perp$: this means that node i has not received any message from j's `mac-broadcast`. This implies that either `mac-broadcast` by j is not yet completed, or node j has not invoked any `mac-broadcast`. In both cases, no STORE by j precedes c.
- $V(j) = v$: by construction, v is in $V(j)$ because STORE$_j(v)$ is invoked before c completes. Next we show that there is no other STORE by node j that completes between two events: the response event of STORE$_j(v)$ and the invocation of c. Assume by way of contradiction that STORE$_j(v')$ completes between these two events. Now observe that: (i) By definition, STORE$_j(v)$ precedes STORE$_j(v')$, so v' is more recent than v from the perspective of node j; and (ii) By the assumption of the abstract MAC layer, when STORE$_j(v')$ completes,

node i must have received the value v'. These two observations together imply that node i will add v' into its view at line 8 before the invocation of c. Consequently, $V(j) = v'$ in the view returned by c, a contradiction.

Property II. Suppose c_1 and c_2 are two COLLECT's such that c_1 returns view V_1, c_2 returns view V_2, and c_1 precedes c_2. By assumption, when `mac-broadcast` completes, all the nodes that have not crashed yet have received the broadcast message. Therefore, $V_1 \preceq V_2$. ◀

From Store-Collect to Linearizable Objects. Constructions of several linearizable shared objects over store-collect are presented in [8]. These constructions only use STORE and COLLECT without relying on other assumptions; hence, can be directly applied on top of MAC-SC. More concretely, Attiya et al. [8] consider a dynamic message-passing system, where nodes continually enter and leave. Similar to our model, their constructions do not assume any information on other participating nodes. All the necessary coordination is through the store-collect object.

This stacked approach sheds light on the computability of the abstract MAC layer. We can use the approach in [8] to implement an atomic register on top of MAC-SC in abstract MAC layer in a *wait-free* manner. Consequently, MAC-SC opens the door for the implementation of many shared objects with consensus number 1. In particular, any implementation on atomic register that does not require a priori information on participating nodes can be immediately applied, e.g., linearizable abort flags, sets, and max registers [26, 8].

Interestingly, despite the strong guarantee, *not all* objects with consensus number 1 can be implemented in the abstract MAC layer. In particular, we prove that $(n-1)$-set consensus is impossible to achieve in our technical report [42]. Our proof follows the structure of the counting-based argument developed by Attiya and Paz (for the shared memory model) [9].

4 Anonymous Storage-Efficient Randomized Binary Consensus

While general, the stacked approach comes with two drawbacks in practice – assumption of unique identities and high storage complexity. Stacking prior shared-memory algorithms on top of MAC-SC requires $\Omega(n \log n)$ due to the usage of store-collect. Prior message-passing algorithms (e.g., [19, 38, 11]) usually require the assumption of unique identities.

This section considers anonymous storage-efficient randomized binary consensus. Recall that deterministic consensus is impossible under our assumptions [34], so the randomized version is the best we can achieve. As shown in Table 1, the state-of-the-art algorithm NR18 [33] requires $O(n^3 \log n)$ time complexity w.h.p. and $\Theta(n \log n)$ storage complexity. We present two *anonymous wait-free* algorithms using only constant storage complexity.

Our Techniques. Our algorithms are inspired by Aspnes's framework [3] of alternating adopt-commit objects and conciliator objects. The framework is designed for the shared memory model, requiring *both* node identity and the knowledge of n. Moreover, it requires $O(\log n)$ atomic multi-writer registers in expectation.

To address these limitations, we have two key technical contributions. First, we replace atomic multi-writer registers by `mac-broadcast`, while using only constant storage complexity. Second, we integrate the "doubling technique," for estimating the system size n, with the framework and present an accompanied analysis to bound the expected round complexity.

More concretely, we combine the technique from [37] and Aspnes's framework to avoid using new objects in a new phase. More precisely, we borrow the "jump" technique from [37], which allows nodes to skip phases (and related messages), to reduce storage complexity. This

Algorithm 2 MAC-AdoptCommit Algorithm: Steps at each node i with input v_i.

```
Local Variables: /* These can be accessed by any thread at $i$. */
    $seen_i[0]$                                  ▷Boolean, initialized to $false$
    $seen_i[1]$                                  ▷Boolean, initialized to $false$
    $proposal_i$                                 ▷value, initialized to $\bot$

mac-broadcast(VALUE, $v_i$)
if $proposal_i \neq \bot$ then
    $v_i \leftarrow proposal_i$
mac-broadcast(PROPOSAL, $v_i$)
if $seen_i[-v_i]$ = false then
    return $(commit, v_i)$
else
    return $(adopt, v_i)$

// Background message handler
Upon receive(VALUE, $v$) do
    $seen_i[v] \leftarrow true$

Upon receive(PROPOSAL, $v$) do
    $proposal_i \leftarrow v$
```

comes with two technical challenges. First, our proofs are quite different from the one in [37], because nodes progress in a different dynamic due to the characteristics of the abstract MAC layer. In particular, we need to carefully analyze which broadcast message has been processed to ensure that the nodes are in the right phase in our proof. This is also where we need to rely on Remark 1, which is usually not needed in the proofs for point-to-point message-passing models. Second, compared to [3], our proofs are more subtle in the sense that we need to make sure that concurrent broadcast events and "jumps" do not affect the probability analysis. The proof in [3] mainly relies on the atomicity of the underlying shared memory, whereas our proofs need to carefully analyze the timing of broadcast events. (Recall that we choose not to use MAC-SC, since it requires nodes to have unique identities.)

Prior solutions rely on the knowledge of network size n [14, 15, 3] or an estimation of n [33] to improve time complexity. For anonymous storage-efficient algorithms, nodes do not know n, and there is no unique node identity. The solution for estimating the network size in the abstract MAC layer in [33] only works correctly with a large n (i.e., with high probability). We integrate a "doubling technique" to *locally* estimate n which does not require any message exchange. For our second randomized binary consensus algorithm MAC-RBC2, nodes double the estimated system size n' every c phases for some constant c, if they have not terminated yet. We identify a proper value of c so that n' is within a constant factor of n, and nodes achieve agreement using $O(n \log n)$ broadcasts on expectation.

Randomized Binary Consensus and Adopt-Commit.

▶ **Definition 4** (Randomized Binary Consensus). *A correct randomized binary consensus algorithm satisfies: (i)* ***Probabilistic Termination****: Each fault-free node decides an output value with probability 1 in the limit; (ii)* ***Validity****: Each output is some input value; and (iii)* ***Agreement****: The outputs are identical.*

▶ **Definition 5** (Adopt-Commit Object). *A correct adopt-commit algorithm satisfies: (i)* ***Termination****: Each fault-free node outputs either $(commit, v)$ or $(adopt, v)$ within a finite amount of time; (ii)* ***Validity****: The v in the output tuple must be an input value; (iii)* ***Coherence****: If a node outputs $(commit, v)$, then any output is either $(adopt, v)$ or $(commit, v)$; and (iv)* ***Convergence****: If all inputs are v, then all fault-free nodes output $(commit, v)$.*

4.1 Algorithm MAC-AdoptCommit

We present MAC-AdoptCommit, which implements a *wait-free* adopt-commit object for binary inputs in the abstract MAC layer model. The pseudo-code is presented in Algorithm 2, and the algorithm is inspired by the construction in shared memory [4]. Following the convention, we will use $-v$ to denote the opposite (or complement) value of value v.

Each node i has two Booleans, $seen_i[0]$ and $seen_i[1]$, and a value $proposal_i$. The former variables are initialized to $false$, and used to denote whether a node i has seen input value 0 and 1, respectively. The last variable $proposal_i$ is initialized to $\perp$, and used to record the "proposed" output from some node. The algorithm has two types of messages:

- A `VALUE` type message $(\texttt{VALUE}, v_i)$ that is used to exchange input values.
- A `PROPOSAL` type message $(\texttt{PROPOSAL}, v_i)$ that is to announce a proposed value.

Upon receiving the message $(\texttt{VALUE}, v)$, node i updates $seen_i[v]$ to $true$ (line 11), denoting that it has seen the value v. Upon receiving the message $(\texttt{PROPOSAL}, v)$, i updates $proposal_i$ to v (line 13), denoting that it has recorded the proposed value, by either itself or another node. Due to concurrency and asynchrony, it is possible that there are multiple proposal messages; thus, node i may overwrite existing value in $proposal_i$ with an opposite value.

Node i first broadcasts input v_i. After `mac-broadcast` completes (line 1), i checks whether it has received any `PROPOSAL` message. If so, it updates its state v_i to the value (line 5). Otherwise, it becomes a proposer and broadcast `PROPOSAL` message with its own input v_i (line 3). After Line 5, the state v_i could be i's original input, or a state copied from the proposed value (from another node). Finally, if node i has not observed any `VALUE` message with the opposite state $(-v_i)$, then it outputs $(commit, v_i)$; otherwise, it outputs $(adopt, v_i)$.

Correctness. Validity, termination and convergence are obvious. To see how MAC-AdoptCommit achieves coherence, first observe that it is impossible for some node to output $(commit, v)$, and the others to output $(commit, -v)$. It is due to the property of `mac-broadcast`: if some node outputs $(commit, v)$, then every node must observe $seen[v] = true$ when executing line 6. Second, if a node outputs $(commit, v)$, then it must be the case that there has already been a proposer that has broadcast both message $(\texttt{VALUE}, v)$ and message $(\texttt{PROPOSAL}, v)$. Therefore, it is impossible for a node to output $(adopt, -v)$. For completeness, we present the proof of correctness in Appendix A.

4.2 Algorithm MAC-RBC

We present MAC-RBC in Algorithm 3. The algorithm uses a sequence of adopt-commit and conciliator objects. A conciliator object helps nodes to reach the same state, and an adopt-commit is used to determine whether it is safe to output a value, and choose a value for the next phase when one cannot "commit" to an output. We adapt MAC-AdoptCommit to store phase index, which allows nodes to jump to a higher phase. Effectively, any adopt-commit object with a phase $< p$ can be interpreted as having $\perp$ in phase p. This also allows us to "reuse" the object. For the conciliator object, we use Ben-Or's local coin [11], which achieves expected exponential time complexity.

In Algorithm 3, line 3 to line 10 effectively implement a reusable adopt-commit object using `VALUE` and `PROPOSAL` messages. Line 12 to line 17 implement a conciliator object using the $\texttt{VALUE}^2$ message. The *seen* variables store both a Boolean and a phase index. Nodes only update these variables when receiving a corresponding message from the same or a higher phase. A node i flips a local coin to decide the state for the next phase at line 17 *only if* it can safely infer that both 0 and 1 are some node's state at the beginning of the phase, i.e., it flips a coin when it has *not* seen a $\texttt{VALUE}^2$ message from a higher phase (line 13), and it has observed a $\texttt{VALUE}^2$ message with value $-v_i$ from the same phase (line 16).

Correctness Proof. It is straightforward to see that MAC-RBC satisfies validity, since the state is either one's input or a value learned from received messages (which must be an input value) and there is no Byzantine fault. We then prove the agreement property.

Algorithm 3 MAC-RBC Algorithm: Steps at each node i with input x_i.

```
Local Variables: /* These variables can be accessed and modified by any thread at node $i$. */
    $seen_i[0]$          ▷(Boolean, phase), initialized to $(false, 0)$
    $seen_i[1]$          ▷(Boolean, phase), initialized to $(false, 0)$
    $seen_i^2[0]$        ▷(Boolean, phase), initialized to $(false, 0)$
    $seen_i^2[1]$        ▷(Boolean, phase), initialized to $(false, 0)$
    $v_i$                ▷state, initialized to $x_i$, the input at node $i$
    $p_i$                ▷phase, initialized to 0
    $proposal_i$         ▷(value, phase), initialized to $(\perp, 0)$

 1: while true do
 2:     $p_{old} \leftarrow p_i$
 3:     mac-broadcast(VALUE, $v_i, p_i$)
 4:     if $proposal_i.phase \geq p_i$ then
 5:         $(v_i, p_i) \leftarrow proposal_i$
 6:     mac-broadcast(PROPOSAL, $v_i, p_i$)
 7:     if $p_{old} \neq p_i$ then
 8:         go to line 2 in $p_i$            ▷"Jump" to $p_i$
 9:     else if $seen_i[-v_i].phase < p_i$ then
10:         output $v_i$
11:     else
12:         mac-broadcast(VALUE$^2$, $v_i, p_i$)
13:         if $seen_i^2[-v_i].phase > p_i$ then
14:             $(v_i, p_i) \leftarrow seen_i^2[-v_i]$
15:             go to line 2 in $p_i$        ▷"Jump" to $p_i$
16:         else if $seen_i^2[-v_i] = (true, p_i)$ then
17:             $v_i \leftarrow$ FlipLocalCoin()
18:         $p_i \leftarrow p_i + 1$             ▷"Move" to $p_i$

    // Background message handler
19: Upon receive(VALUE, $v, p$) do
20:     if $p \geq p_i$ then
21:         $seen_i[v] \leftarrow (true, p)$

22: Upon receive(VALUE$^2$, $v, p$) do
23:     if $p \geq p_i$ then
24:         $seen_i^2[v] \leftarrow (true, p)$

25: Upon receive(PROPOSAL, $v, p$) do
26:     if $p \geq p_i$ then
27:         $proposal_i \leftarrow (v, p)$
```

▶ **Lemma 6.** *Suppose node i is the first node that makes an output and it outputs v in phase p, then all the other nodes either output v in phase p or phase $p + 1$.*

Proof. Suppose node i outputs v in phase p at time T_3. Then it must have $seen_i[-v].phase < p$. Assume this holds true at time T_2. Furthermore, assume line 6 was executed at time T_1 by node i at time T_1 such that $T_1 < T_2 < T_3$.

We first make the observation, namely *Obs1*, no node with $-v$ in phase $p' \geq p$ has completed line 3 at time $\leq T_2$. Suppose node j has state $-v$ in some phase $p' \geq p$. By assumption (in Section 2), before node i starts to execute line 9 at time T_2, its message handler has processed all the messages received by the abstract MAC layer. Therefore, the fact that $seen_i[-v].phase < p_i$ at time T_2 implies that node i has *not* receive any message of the form (VALUE, $-v, p'$) at time T_2. Consequently, node j has not completed mac-broadcast(VALUE, $-v, p'$) (line 1) at time T_2.

Consider the time T when the first mac-broadcast(VALUE, $-v, p$) is completed (if there is any). At time T, there are two cases for node k that has not crashed yet:

- Node k has *not* moved beyond phase p:
  k must have already received (PROPOSAL, v) at some earlier time than T, because (i) *Obs1* implies that $T > T_2$; and (ii) by time T, node i has already completed line 6 (which occurred at time T_1). Consider two scenarios: (s1) k executes line 4 after receiving (PROPOSAL, v): k sets $proposal_k$ to value v before executing line 6 (potentially at some later point than T); and (s2) k executes line 4 before receiving (PROPOSAL, v): in this case: k's input at phase p must be v; otherwise, T cannot be the first mac-broadcast(VALUE, $-v, p'$) that is completed. (Observe that by assumption of this case, k executes line 4 before node i completes its line 6 at time T_1.)
- Node k has moved beyond phase p:
  By assumption, time T is the time that the first mac-broadcast(VALUE, $-v, p$) is completed. Thus, it is impossible for node k to have set (v_k, p_k) to $(-v, p')$ for some $p' \geq p$.

In both cases, right before executing line 6, node k can only `mac-broadcast(PROPOSAL,` v, p'`)`, for $p' \geq p$, i.e., no `mac-broadcast(PROPOSAL,` $-v, p'$`)` is possible. Consequently, the lemma then follows by a simple induction on the order of nodes moving to phase $p+1$. ◀

Since we assume a message oblivious adversary, the termination and exponential time complexity follow the standard argument of using local coins [11]. In particular, we have the following Theorem, which implies that MAC-RBC requires, on expectation, an exponential number of broadcasts. The proof is deferred to Appendix B.

▶ **Theorem 7.** *For any $\delta \in (0,1)$, let $p = \lceil 2^{n-1} \ln(1/\delta) \rceil$. Then with probability at least $1-\delta$, MAC-RBC terminates within p phases. (In other words, all nodes have phase $\leq p$.)*

4.3 MAC-RBC2: Improving Time Complexity

There are several solutions for an efficient conciliator object, such as a shared coin [5] and the "first-mover-win" strategy [14, 15, 3]. The first-mover-win strategy was developed for a single multi-writer register in shared memory such that agreement is achieved when only one winning node (the first mover) successfully writes to the register. If there are concurrent operations, then agreement might be violated. On a high-level, this strategy translates to the "first-broadcaster-win" design in the abstract MAC layer. One challenge in our analysis is the lack of the atomicity of the register. We need to ensure that even in the presence of concurrent broadcast and failure events, there is still a constant probability for achieving agreement, after nodes have a "good enough" estimated system size n'.

Conciliator and Integration. Our conciliator object is presented in Algorithm 4, which is inspired by the ImpatientFirstMover strategy [3]. The key difference from [3] is that MAC-FirstMover uses an estimated size n', instead of the actual network size n (as in [3]), which makes the analysis more complicated, as our analysis depends on both n and n'. Algorithm 4 presents a standalone conciliator implementation. We will later describe how to integrate it with Algorithm 3 by adding the field of phase index and extra message handlers.

In our design, each node proceeds in rounds and increases the probability of revealing their coin-flip after each round k, if it has not learned any coin flip at line 2. To prevent the message adversary from scheduling concurrent messages with conflicting values, nodes have two types of messages: `COIN` and `DUMMY`. The first message is used to reveal node's input v_i, whereas the second is used as a "decoy" that has no real effect. At line 3, node i draws a local random number between $[0,1)$ to decide which message to broadcast. Since the adversary is oblivious, it cannot choose its scheduling based on the message type.

MAC-RBC2 can be obtained by integrating Alg. 4 (MAC-FirstMover) with Alg. 3 (MAC-RBC) with the changes below. The complete algorithm is presented in Appendix C.

- FlipLocalCoin() is replaced by MAC-FirstMover($2^{\lfloor \frac{p_i}{c} \rfloor} n_0$), where c is a constant to be defined later and n_0 is a constant that denotes the initial guess of the system size. All nodes have an identical information of c and n_0 in advance. Therefore, nodes in the same phase call MAC-FirstMover with the same estimated system size n'. Recall that p_i is the phase index local at node i. Hence, effectively in our design, each node i is doubling the estimated size n' every c phases.
- To save space, $coin_i$ consists of two fields $(value, phase)$, and is used in a fashion similar to how $proposal_i$ is used in MAC-RBC. That is, if $coin_i$ has a phase field lower than the current phase p_i, then the value field is treated as $\perp$.

Algorithm 4 MAC-FirstMover Algorithm: Steps at each node i with input v_i.

```
Local Variables: /* These variables can be accessed by any thread at node i. */
    $coin_i$                                    ▷value, initialized to $\bot$
Input: $n'$                      ▷estimated system size, given as an input to MAC-FirstMover

1:  $k \leftarrow 0$
2:  while $coin_i = \bot$ do
3:      if a local random number $< \frac{2^k}{2n'}$ then
4:          mac-broadcast(COIN, $v_i$)
5:      else
6:          mac-broadcast(DUMMY)
7:      $k \leftarrow k + 1$
8:  mac-broadcast(COIN, $coin_i$)
9:  return $coin_i$

    // Background message handler
10: Upon receive(COIN, $v$) do
11:     if $coin_i = \bot$ then
12:         $coin_i \leftarrow v$

13: Upon receive(DUMMY) do
14:     do nothing
```

- The messages by node i are tagged with its current phase p_i. That is, the two messages in Algorithm 4 have the following form: $(\texttt{COIN}, v, p_i)$ and $(\texttt{DUMMY}, p_i)$.
- MAC-RBC2 needs to have two extra message handlers to process DUMMY and COIN messages. The COIN message handler only considers messages with phase $\leq p_i$.
- In MAC-RBC2, nodes jump to a higher phase upon receiving a coin broadcast. More precisely, if a node i receives a coin broadcast m from a phase $p > p_i$, then i updates v_i to the value in m and jumps to phase $p + 1$.

Correctness and Time Complexity. Correctness follows from the prior correctness proof, as MAC-FirstMover is a valid conciliator object that returns only 0 or 1. To analyze time complexity, we start with several useful notions.

▶ **Definition 8** (Active Nodes). *We say a node is an active node in phase p if it ever executes MAC-FirstMover in phase p. Let $\mathcal{A}_p$ denote the set of all active nodes in phase p.*

Due to asynchrony, different nodes might execute MAC-FirstMover in phase p at different times. Moreover, nodes may "jump" to a higher phase in our design. Consequently, not all nodes would execute MAC-FirstMover in phase p for every p.

▶ **Definition 9** (Broadcast). *We distinguish different types of broadcasts, which will later be useful for our probability analysis:*

- *A broadcast is a phase-p broadcast if it is tagged with phase p. By definition, only active nodes in phase p (i.e., nodes in $\mathcal{A}_p$) make phase-p broadcasts.*
- *A broadcast made in MAC-FirstMover (Algorithm 4) is a coin broadcast if its message has the COIN tag; otherwise, it is a dummy broadcast.*
- *A broadcast is an original broadcast if it is made in the while loop (Line 4 and Line 6 in Algorithm 4). It is a follow-up broadcast if it is made after $coin_i$ becomes non-empty (line 8 of Algorithm 4). By design, a follow-up broadcast must be a coin broadcast.*
- *Consider an original broadcast $m = (\texttt{COIN}, v, p)$ by node i. The broadcast is said to be successful in phase p if there exists a node j that completes a follow-up broadcast with $coin_j = v$ in phase p, i.e., node j receives the acknowledgement for its broadcast at line 8 of Algorithm 4. Note that i may not equal to j, and both i and j might be faulty (potentially crash at a future point of time).*

Recall that we define a broadcast to be "*completed*" if a node making the broadcast receives the acknowledge from the abstract MAC layer. This notion should not be confused with the notion of "successful." In particular, we have (i) a broadcast might be completed, but not successful – this is possible if there are multiple original coin broadcasts with different v; (ii) a broadcast might be successful, but not completed – this is possible if a node j receives an original coin broadcast by a faulty node and node j completes the follow-up broadcast.

We will apply the following important observation in our proofs. The observation directly follows from our definition of different broadcasts.

▶ Remark 10. If there is a completed original coin broadcast in phase p, then there must be at least one successful original coin broadcast in phase p.

▶ **Definition 11.** *A node completes MAC-FirstMover of phase p if it receives a coin broadcast of the form $(\texttt{COIN}, *, p')$ with $p' \geq p$.*[2] *Let $\mathcal{C}_p$ denote the set of all nodes that complete MAC-FirstMover of phase p.*

By definition, a node *not* in $\mathcal{A}_p$ can still complete MAC-FirstMover of phase p, if it receives a coin broadcast from a higher phase.

We first bound the number of expected original broadcasts in order for nodes to complete MAC-FirstMover. Recall that k in Algorithm 4 denotes the round index. In our analysis below, we only bound the number of broadcasts made by *fault-free* nodes.

▶ **Lemma 12.** *With probability $\geq 1 - \delta$, all fault-free nodes complete MAC-FirstMover in phase p, after $\leq 2n' \ln(1/\delta)$ original broadcasts are made by fault-free nodes in phase p.*

Proof. We begin with the following claim. It follows from the definition of successful coin broadcasts. For completeness, we include the proof in Appendix D.

▷ Claim 13. All fault-free nodes complete MAC-FirstMover of phase p if there exists *at least one* successful coin broadcast in phase p.

Every broadcast in phase p has probability $\geq \frac{1}{2n'}$ to be a coin broadcast (by line 3 of Algorithm 4). Since we only care about the number of original broadcasts made by fault-free nodes, all these broadcasts must be eventually completed. Consequently, for all the fault-free nodes in $\mathcal{A}_p$, we have the probability that *first t completed broadcasts by any fault-free node in $\mathcal{A}_p$ are all dummy*, denoted by P, bounded by

$$P \leq \prod_{i=1}^{t} \left(1 - \frac{1}{2n'}\right) \leq \exp\left(-\frac{t}{2n'}\right).$$

Equivalently, for any $t \geq 2n' \ln(1/\delta)$, with probability at least $1 - \delta$, there exists at least one completed coin broadcast among the first t completed broadcasts in phase p, which further implies the existence of at least one successful broadcast by Remark 10. This, together with Claim 13, conclude the proof. (Note that there could be a successful coin broadcast by a faulty node in $\mathcal{A}_p$, but this does not affect the lower bound we derived.) ◀

▶ **Lemma 14.** *Consider the case when all active nodes in phase p (i.e., nodes in $\mathcal{A}_p$) execute MAC-FirstMover of phase p with parameter $n' \geq n$. With probability ≥ 0.05, each node $j \in \mathcal{C}_p$ must reach the same state v_j in either phase p or phase $p + 1$.*

Proof. We begin with the following claim. The proof is presented in Appendix E.

▷ Claim 15. If there is exactly one successful original coin broadcast in phase p, then all nodes in $\mathcal{C}_p$ must achieve the same state in either phase p or phase $p + 1$.

[2] This coin broadcast can be an original or a follow-up coin broadcast.

The analysis below aims to identify the lower bound on the probability of the event that there exists exactly one successful original coin broadcast in phase p.

Consider any message scheduling by the adversary. Since we assume it is oblivious, we can define r_i as the probability that the i-th completed original broadcast in phase p, across the entire set of nodes in $\mathcal{A}_p$, is a coin broadcast given this unknown message scheduling. That is, since the schedule by the adversary is chosen a priori, r_i is a fixed number. Next, we introduce two variables:

- Let $T-1$ denote the number of completed original dummy broadcasts in phase p before the first *completed* original coin broadcasts in phase p, given the message scheduling; and
- Let k_j denote the number of completed original dummy broadcasts by a node $j \in \mathcal{A}$, among these $T-1$ broadcasts. Note that only k_j is defined with respect to a single node.

The first definition implies that the T-th completed original broadcast is a coin broadcast.

Without loss of generality, assume that in the given schedule, the i-th completed original broadcasts across the entire set of nodes in $\mathcal{A}_p$ is the k-th completed original broadcast made by node j. Then by Line 3 of Algorithm 4, we can quantify r_i as follows:

$$r_i = \frac{2^{k-1}}{2n'} \tag{1}$$

Observe that if some node $j \in \mathcal{A}_p$ fails to complete an original broadcast, then it cannot make any further broadcasts. This is because if j is not able to complete a broadcast, then it must be a faulty node. Consequently, the k-th "*completed*" original broadcast made by node j must also be the k-th original broadcast by j. Hence, Equation (1) still holds for a faulty j.

Define $t^* = \min\{t : \sum_{i=1}^{t} r_i \geq \frac{1}{4}\}$. Then we have

$$\mathbb{P}\{T > t^*\} = \prod_{i=1}^{t^*}(1 - r_i) \leq \exp\left(-\sum_{i=1}^{t^*} r_i\right) \leq \exp(-1/4). \tag{2}$$

Define $\mathcal{A}'_p$ as the nodes in $\mathcal{A}_p$ that have completed at least one original dummy broadcast among the first $T-1$ completed original dummy broadcasts in phase p. In other words, $j \in \mathcal{A}'_p$ iff $k_j \geq 1$. Then we can derive the following equality, based on the nodes that have made the completed original broadcast(s):

$$\sum_{i=1}^{T-1} r_i = \sum_{j \in \mathcal{A}'_p} \sum_{k=1}^{k_j} \frac{2^{k-1}}{2n'} = \sum_{j \in \mathcal{A}'_p} \frac{2^{k_j} - 1}{2n'}. \tag{3}$$

The first equality follows from the definition that the first $T-1$ broadcasts are all dummy, and thus r_i must "correspond" to the k-th completed original broadcast (for some $1 \leq k \leq k_j$) by some node j, whose prior broadcasts are all dummy as well. Furthermore, the k_j-th completed original dummy broadcast is the last one by node j (among the first $T-1$ broadcasts across the system). Note that by definition, r_i is a constant for all i. However, the summation $\sum_{i=1}^{T-1} r_i$ is indeed a random variable whose randomness comes from each coin flip. This explains why the first equality is valid.

Next, we upper bound the probability that there are multiple original *coin* broadcasts in one phase. Note that every active node in $\mathcal{A}_p$ can make *at most one* original coin broadcast in phase p because a node that makes a original coin broadcast must receive that coin broadcast from itself and thus terminate Algorithm 4. Since by definition, the T-th completed original broadcast is the first completed original coin broadcast in the entire system, any original coin broadcast made by some node $j \in \mathcal{A}_p$ must be the $(k_j + 1)$-th original broadcast by node j. Equation (1) implies that the probability of the $(k_j + 1)$-th original broadcast being a coin broadcast is $\frac{2^{k_j}}{2n'}$.

Let E_p denote the event that there are strictly more than one original coin broadcast in phase p – these coin broadcasts may or may not be successful. Let E_p^c denote its complement. By union bound, we have

$$\mathbb{P}\{E_p\} \leq \sum_{j \in \mathcal{A}_p} \mathbb{P}\{\text{node } j \text{ makes an original coin broadcast}\} = \sum_{j \in \mathcal{A}_p} \frac{2^{k_j}}{2n'}.$$

Consequently, by Equation (3), the definition of t^* such that $\sum_{i=1}^{t} r_i < \frac{1}{4}$ for all $t < t^*$, and the assumption that $n' \geq n \geq |\mathcal{A}_p|$, we have

$$\begin{aligned}
\mathbb{P}\{E_p \,|\, T \leq t^*\} &\leq \sum_{j \in \mathcal{A}_p} \frac{2^{k_j}}{2n'} = \sum_{j \in \mathcal{A}'_p} \frac{2^{k_j}}{2n'} + \sum_{j \in \mathcal{A}_p - \mathcal{A}'_p} \frac{1}{2n'} \qquad (k_j = 0 \text{ for } j \notin \mathcal{A}'_p) \\
&= \left(\sum_{j \in \mathcal{A}'_p} \frac{2^{k_j} - 1}{2n'} + \sum_{j \in \mathcal{A}'_p} \frac{1}{2n'} \right) + \sum_{j \in \mathcal{A}_p - \mathcal{A}'_p} \frac{1}{2n'} \\
&= \sum_{i=1}^{T-1} r_i + \sum_{j \in \mathcal{A}_p} \frac{1}{2n'} = \sum_{i=1}^{T-1} r_i + \frac{|\mathcal{A}_p|}{2n'} \leq \frac{3}{4}.
\end{aligned}$$

By Remark 10, $T \leq t^*$, which denotes the event that there is at least one completed original coin broadcast in the first t^* completed original broadcasts, implies that there is at least one *successful* original broadcast in the first t^* completed original broadcasts. Therefore, the fact that $T \leq t^*$ together with E_p^c is a *subset* of the events that there is exactly one successful original coin broadcast in phase p. Consequently, together with Equation (2), we have

$$\begin{aligned}
&\mathbb{P}\{\text{exactly one successful original coin broadcast in phase } p\} \\
&\geq \mathbb{P}\{E_p^c, T \leq t^*\} \geq (1 - \exp(-1/4))(1 - 3/4) \geq 0.05.
\end{aligned}$$

This combined with Claim 15 prove the lemma. ◀

Define the constant c as follows: $c = \frac{\ln(2/\delta)}{0.05}$. Using c in MAC-FirstMover (Algorithm 4), we can derive the following theorem. The full proof is presented in Appendix F. Roughly speaking, nodes need $O(\log n)$ phases to have a large enough estimated system size n'. After that, nodes need a constant number of phases to reach agreement and terminate, due to Lemma 14. Next, Lemma 12 states that each phase needs $O(n)$ broadcasts on expectation. These give us the desired result.

▶ **Theorem 16.** *With probability $\geq 1 - \delta$, MAC-RBC2 terminates and achieves agreement using $O(n \log n)$ total broadcasts across the entire system.*

References

1 Mohssen Abboud, Carole Delporte-Gallet, and Hugues Fauconnier. Agreement without knowing everybody: a first step to dynamicity. In Djamal Benslimane and Aris M. Ouksel, editors, *Proceedings of the 8th international conference on New technologies in distributed systems, NOTERE '08, Lyon, France, June 23-27, 2008*, pages 49:1–49:5. ACM, 2008. doi:10.1145/1416729.1416792.

2 Eduardo Adílio Pelinson Alchieri, Alysson Neves Bessani, Joni da Silva Fraga, and Fabíola Greve. Byzantine consensus with unknown participants. In Theodore P. Baker, Alain Bui, and Sébastien Tixeuil, editors, *Principles of Distributed Systems, 12th International Conference, OPODIS 2008, Luxor, Egypt, December 15-18, 2008. Proceedings*, volume 5401 of *Lecture Notes in Computer Science*, pages 22–40. Springer, 2008. doi:10.1007/978-3-540-92221-6_4.

3 James Aspnes. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. *Distributed Comput.*, 25(2):179–188, 2012. doi:10.1007/s00446-011-0134-8.

4 James Aspnes and Faith Ellen. Tight bounds for anonymous adopt-commit objects. In Rajmohan Rajaraman and Friedhelm Meyer auf der Heide, editors, *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 317–324. ACM, 2011. doi:10.1145/1989493.1989548.

5 James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous pram model. In *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, pages 340–349, 1990. doi:10.1145/97444.97701.

6 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995. doi:10.1145/200836.200869.

7 Hagit Attiya, Arie Fouren, and Eli Gafni. An adaptive collect algorithm with applications. *Distributed Comput.*, 15(2):87–96, 2002. doi:10.1007/s004460100067.

8 Hagit Attiya, Sweta Kumari, Archit Somani, and Jennifer L. Welch. Store-collect in the presence of continuous churn with application to snapshots and lattice agreement. In Stéphane Devismes and Neeraj Mittal, editors, *Stabilization, Safety, and Security of Distributed Systems - 22nd International Symposium, SSS 2020, Austin, TX, USA, November 18-21, 2020, Proceedings*, volume 12514 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2020. doi:10.1007/978-3-030-64348-5_1.

9 Hagit Attiya and Ami Paz. Counting-based impossibility proofs for renaming and set agreement. In Marcos K. Aguilera, editor, *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*, volume 7611 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2012. doi:10.1007/978-3-642-33651-5_25.

10 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* Wiley Series on Parallel and Distributed Computing, 2004.

11 Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 27–30, New York, NY, USA, 1983. ACM. doi:10.1145/800221.806707.

12 François Bonnet and Michel Raynal. Anonymous asynchronous systems: the case of failure detectors. *Distributed Comput.*, 26(3):141–158, 2013. doi:10.1007/s00446-012-0169-5.

13 David Cavin, Yoav Sasson, and André Schiper. Consensus with unknown participants or fundamental self-organization. In Ioanis Nikolaidis, Michel Barbeau, and Evangelos Kranakis, editors, *Ad-Hoc, Mobile, and Wireless Networks: Third International Conference, ADHOC-NOW 2004, Vancouver, Canada, July 22-24, 2004. Proceedings*, volume 3158 of *Lecture Notes in Computer Science*, pages 135–148. Springer, 2004. doi:10.1007/978-3-540-28634-9_11.

14 Ling Cheung. Randomized wait-free consensus using an atomicity assumption. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, volume 3974 of *Lecture Notes in Computer Science*, pages 47–60. Springer, 2005. doi:10.1007/11795490_6.

15 Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM J. Comput.*, 23(4):701–712, 1994. doi:10.1137/S0097539790192635.

16 Eli Daian, Giuliano Losa, Yehuda Afek, and Eli Gafni. A wealth of sub-consensus deterministic objects. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 17:1–17:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.DISC.2018.17.

17 Giuseppe Di Luna and Roberto Baldoni. Non Trivial Computations in Anonymous Dynamic Networks. In Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Butucaru, editors,

19th International Conference on Principles of Distributed Systems (OPODIS 2015), volume 46 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33:1–33:16, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.OPODIS.2015.33.

18 Giuseppe Antonio Di Luna and Roberto Baldoni. Brief announcement: Investigating the cost of anonymity on dynamic networks. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 339–341, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2767386.2767442.

19 Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33:499–516, May 1986. doi:10.1145/5925.5931.

20 Mohsen Ghaffari, Erez Kantor, Nancy A. Lynch, and Calvin C. Newport. Multi-message broadcast with abstract MAC layers and unreliable links. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 56–65. ACM, 2014. doi:10.1145/2611462.2611492.

21 Fabíola Greve and Sébastien Tixeuil. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings*, pages 82–91. IEEE Computer Society, 2007. doi:10.1109/DSN.2007.61.

22 D. Hendler, F. Fich, and N. Shavit. Linear lower bounds on real-world implementations of concurrent objects. In *Proc. 46th Annual IEEE Symposium on Foundations of Computer Science*, 2005.

23 M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1), January 1991. doi:10.1145/114005.102808.

24 Majid Khabbazian, Dariusz R. Kowalski, Fabian Kuhn, and Nancy A. Lynch. Decomposing broadcast algorithms using abstract MAC layers. *Ad Hoc Networks*, 12:219–242, 2014. doi:10.1016/j.adhoc.2011.12.001.

25 Fabian Kuhn, Nancy A. Lynch, and Calvin C. Newport. The abstract MAC layer. *Distributed Comput.*, 24(3-4):187–206, 2011. doi:10.1007/s00446-010-0118-0.

26 Petr Kuznetsov, Thibault Rieutord, and Sara Tucci Piergiovanni. Reconfigurable lattice agreement and applications. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, volume 153 of *LIPIcs*, pages 31:1–31:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.OPODIS.2019.31.

27 Giuseppe Antonio Di Luna, Roberto Baldoni, Silvia Bonomi, and Ioannis Chatzigiannakis. Conscious and unconscious counting on anonymous dynamic networks. In Mainak Chatterjee, Jiannong Cao, Kishore Kothapalli, and Sergio Rajsbaum, editors, *Distributed Computing and Networking - 15th International Conference, ICDCN 2014, Coimbatore, India, January 4-7, 2014. Proceedings*, volume 8314 of *Lecture Notes in Computer Science*, pages 257–271. Springer, 2014. doi:10.1007/978-3-642-45249-9_17.

28 Giuseppe Antonio Di Luna, Roberto Baldoni, Silvia Bonomi, and Ioannis Chatzigiannakis. Counting in anonymous dynamic networks under worst-case adversary. In *IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014*, pages 338–347. IEEE Computer Society, 2014. doi:10.1109/ICDCS.2014.42.

29 Giuseppe Antonio Di Luna and Giovanni Viglietta. Brief announcement: Efficient computation in congested anonymous dynamic networks. In Rotem Oshman, Alexandre Nolin, Magnús M. Halldórsson, and Alkida Balliu, editors, *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC 2023, Orlando, FL, USA, June 19-23, 2023*, pages 176–179. ACM, 2023. doi:10.1145/3583668.3594590.

30 Giuseppe Antonio Di Luna and Giovanni Viglietta. Optimal computation in leaderless and multi-leader disconnected anonymous dynamic networks. In Rotem Oshman, editor, *37th International Symposium on Distributed Computing, DISC 2023, October 10-12, 2023,*

L'Aquila, Italy, volume 281 of *LIPIcs*, pages 18:1–18:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.DISC.2023.18.

31 Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

32 Nancy A. Lynch, Tsvetomira Radeva, and Srikanth Sastry. Asynchronous leader election and MIS using abstract MAC layer. In Fabian Kuhn and Calvin C. Newport, editors, *FOMC'12, The Eighth ACM International Workshop on Foundations of Mobile Computing (part of PODC 2012), Funchal, Portugal, July 19, 2012, Proceedings*, page 3. ACM, 2012. doi:10.1145/2335470.2335473.

33 Calvin Newport and Peter Robinson. Fault-tolerant consensus with an abstract MAC layer. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 38:1–38:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.DISC.2018.38.

34 Calvin C. Newport. Consensus with an abstract MAC layer. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 66–75. ACM, 2014. doi:10.1145/2611462.2611479.

35 Eric Ruppert. The anonymous consensus hierarchy and naming problems. In Eduardo Tovar, Philippas Tsigas, and Hacène Fouchal, editors, *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings*, volume 4878 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2007. doi:10.1007/978-3-540-77096-1_28.

36 Lewis Tseng and Callie Sardina. Byzantine Consensus in Abstract MAC Layer. In Alysson Bessani, Xavier Défago, Junya Nakamura, Koichi Wada, and Yukiko Yamauchi, editors, *27th International Conference on Principles of Distributed Systems (OPODIS 2023)*, volume 286 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:16, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.OPODIS.2023.9.

37 Lewis Tseng, Qinzi Zhang, and Yifan Zhang. Brief announcement: Reaching approximate consensus when everyone may crash. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 53:1–53:3. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.53.

38 Nitin H. Vaidya, Lewis Tseng, and Guanfeng Liang. Iterative approximate Byzantine consensus in arbitrary directed graphs. In *Proceedings of the thirty-first annual ACM symposium on Principles of distributed computing*, PODC '12. ACM, 2012. doi:10.1145/2332432.2332505.

39 Dongxiao Yu, Yifei Zou, Yuexuan Wang, Jiguo Yu, Xiuzhen Cheng, and Francis C. M. Lau. Implementing the abstract MAC layer via inductive coloring under the rayleigh-fading model. *IEEE Trans. Wirel. Commun.*, 20(9):6167–6178, 2021. doi:10.1109/TWC.2021.3072236.

40 Dongxiao Yu, Yifei Zou, Jiguo Yu, Yong Zhang, Feng Li, Xiuzhen Cheng, Falko Dressler, and Francis C. M. Lau. Implementing the abstract MAC layer in dynamic networks. *IEEE Trans. Mob. Comput.*, 20(5):1832–1845, 2021. doi:10.1109/TMC.2020.2971599.

41 Dongxiao Yu, Yifei Zou, Yong Zhang, Hao Sheng, Weifeng Lv, and Xiuzhen Cheng. An exact implementation of the abstract MAC layer via carrier sensing in dynamic networks. *IEEE/ACM Trans. Netw.*, 29(3):994–1007, 2021. doi:10.1109/TNET.2021.3057890.

42 Qinzi Zhang and Lewis Tseng. The power of abstract mac layer: A fault-tolerance perspective, 2024. arXiv:2408.10779.

A Correctness Proof of MAC-AdoptCommit

▶ **Theorem 17.** *MAC-AdoptCommit is correct for binary inputs.*

Proof. MAC-AdoptCommit satisfies validity, because v_i is either an input at node i or a value from $proposal_i$, which must be an input from another node.

MAC-AdoptCommit satisfies termination, because all the steps are non-blocking.

MAC-AdoptCommit satisfies coherence. Suppose node i outputs $(commit, v)$ at time T_3, and completes line 5 at T_2, and line 4 at T_1 such that $T_1 < T_2 < T_3$.

We first make the following observation, namely *Obs1*, no node with input $-v$ has completed line 1 at any time $\leq T_2$. Suppose node j has input $-v$. By Remark 1 in Section 2, before node i starts to execute line 5, its message handler has processed all the messages received by the abstract MAC layer. Therefore, the fact that $seen_i[-v]$ = false at time T_2 implies that node i has *not* receive any message of the form $(\texttt{VALUE}, -v)$ at time T_2. Consequently, node j has not completed `mac-broadcast`(`VALUE`, $-v$) (line 1) at time T_2.

Consider the time T when the first `mac-broadcast`(`VALUE`, $-v$) is completed (if there is any). At time T, any node k that has not crashed yet must have already received (`PROPOSAL`, v) at some earlier time than T, because (i) *Obs1* implies that $T > T_2$; and (ii) by time T, node i has already completed line 4 (which occurred at time T_1). Consider two cases:

- k executes line 2 after receiving (`PROPOSAL`, v): in this case, k sets $proposal_k$ to value v before executing line 3 (potentially at some later point that T).
- k executes line 2 before receiving (`PROPOSAL`, v): in this case: k's input must be v; otherwise, T cannot be the first `mac-broadcast`(`VALUE`, $-v$) that is completed. (Observe that by assumption of this case, k executes line 2 before node i completes its line 4 at time T_1.)

In both cases, at line 4, node k can only `mac-broadcast`(`PROPOSAL`, v). That is, no `mac-broadcast`(`PROPOSAL`, $-v$) is possible. Consequently, coherence is satisfied.

MAC-AdoptCommit satisfies convergence. If all the inputs are v, then the only value that can appear in $proposal_i$ is v for each node i. Moreover, none of the nodes would broadcast $-v$; hence, $seen_i[-v]$ will always be false. Consequently, all nodes would output $(commit, v)$. ◀

B Proof of Theorem 7

Proof. Recall that we assume the message oblivious adversary; hence, termination proof is more straightforward. This is because if no node outputs a value, then all nodes rely on the conciliator (flipping a local coin) to reach the same states for the next phase. By construction, nodes may (i) jump to a higher phase with a copied state, (ii) obtain a state that is equivalent to the proposed value from a `PROPOSAL` message, or (iii) choose its new state randomly. Therefore, there is a non-zero probability that all of these random choices equal to the unique state value obtained using approach (i) or (ii). The reason that these obtained states are identical is due to the *coherence* property of the adopt-commit object (as proved in Appendix A).

In the worst case, all nodes "move in sync," i.e., they enter the same phase concurrently without using the jump, and have their states randomly generated. Otherwise if there is some "fast" node that is in a higher phase, it may force all other nodes to jump to its state after it becomes the "proposer" at line 6. We denote the probability that all states are equal after flipping a local coin by r^*. Clearly, $r^* = 2^{-(n-1)} > 0$. Let P be the random variable that denotes the termination phase of MAC-RBC, and note that $P > p$ only if the states are not equal in the first p rounds. Therefore, $\mathbb{P}\{P > p\} \le (1 - r^*)^p$. Finally, we conclude the proof by showing that for all $p \ge \ln(1/\delta)/r^* = 2^{n-1}\ln(1/\delta)$,

$$(1 - r^*)^p \le (1 - r^*)^{\ln(1/\delta)/r^*} \le \exp(-\ln(1/\delta)) = \delta.$$

The inequality follows from the identity that $1 - x \le \exp(-x)$ for all $x > 0$. ◀

C MAC-RBC2

Algorithm 5 MAC-RBC2 Algorithm: Steps at each node i with input x_i.

```
Local Variables: /* These variables can be accessed and modified by any thread at node $i$. */
    $seen_i[0]$        ▷(Boolean, phase), initialized to $(false, 0)$
    $seen_i[1]$        ▷(Boolean, phase), initialized to $(false, 0)$
    $seen^2_i[0]$      ▷(Boolean, phase), initialized to $(false, 0)$
    $seen^2_i[1]$      ▷(Boolean, phase), initialized to $(false, 0)$
    $v_i$              ▷state, initialized to $x_i$, the input at node $i$
    $p_i$              ▷phase, initialized to 0
    $proposal_i$       ▷(value, phase), initialized to $(\perp, 0)$
    $n_0$              ▷an initial guess of system size, initialized to some constant natural number
    $n'$               ▷estimated system size, initialized to 1
    $c$                ▷a constant defined as $c = \frac{\ln(2/\delta)}{0.05}$
    $coin_i$           ▷(Boolean, phase), initialized to $(\perp, -1)$

 1: mac-broadcast(ID, $i$)
 2: while true do
 3:     $p_{old} \leftarrow p_i$
 4:     mac-broadcast(VALUE, $v_i, p_i$)
 5:     if $proposal_i.phase \ge p_i$ then
 6:         $(v_i, p_i) \leftarrow proposal_i$
 7:     mac-broadcast(PROPOSAL, $v_i, p_i$)
 8:     if $p_{old} \ne p_i$ then
 9:         go to line 2                          ▷"Jump" to $p_i$
10:     else if $seen_i[-v_i].phase < p_i$ then
11:         output $v_i$
12:         mac-broadcast(VALUE$^2$, $v_i, p_i$)
13:         if $seen^2_i[-v_i].phase > p_i$ then
14:             $(v_i, p_i) \leftarrow (-v_i, seen^2_i[-v_i].phase)$
15:             go to line 2                      ▷"Jump" to $p_i$
16:         else if $seen^2_i[-v_i] = (true, p_i)$ then
17:             // MAC-FirstMover
18:             $n' \leftarrow 2^{\lfloor \frac{p_i}{c} \rfloor} n_0$
19:             $k \leftarrow 0$
20:             while $coin_i.phase < p_i$ do
21:                 if a local random number $< \frac{2^k}{2n'}$ then
22:                     mac-broadcast(COIN, $v_i, p_i$)
23:                 else
24:                     mac-broadcast(DUMMY)
25:                 $k \leftarrow k + 1$
26:             mac-broadcast(COIN, $v, p$)
27:             $(v_i, p_i) \leftarrow coin_i$
28:         $p_i \leftarrow p_i + 1$                   ▷"Move" to $p_i$

    // Background message handlers
29: Upon receive(VALUE, $v, p$) do
30:     if $p \ge seen_i[v].phase$ then
31:         $seen_i[v] \leftarrow (true, p)$

32: Upon receive(VALUE$^2$, $v, p$) do
33:     if $p \ge seen^2_i[v].phase$ then
34:         $seen^2_i[v] \leftarrow (true, p)$

35: Upon receive(PROPOSAL, $v, p$) do
36:     if $p \ge proposal_i.phase$ then
37:         $proposal_i \leftarrow (v, p)$

    // Message handlers for MAC-FirstMover
38: Upon receive(COIN, $v, p$) do
39:     if $p = p_i$ and $p > coin_i.phase$ then
40:         $coin_i \leftarrow (v, p)$
41:     else if $p > p_i$ then
42:         $(v_i, p_i) \leftarrow (v, p + 1)$
43:         go to line 2                          ▷"Jump" to $p_i$

44: Upon receive(DUMMY) do
45:     do nothing
```

We can get rid of the *coin* variable and directly use v_i and p_i. However, we choose to reserve the variable so that it is more obvious how MAC-RBC2 utilizes MAC-FirstMover.

The reasons that we need to have the condition $p > coin_i.phase$ are: (i) $coin_i.phase$ may be decoupled from p_i; and (ii) each node i has at most two coin broadcasts for a phase p.

D Proof of Claim 13

Proof of Claim 13. Let $m = (\texttt{COIN}, v, p)$ be a successful coin broadcast in phase p. Recall that m is successful because there exists a node j that completes the follow-up broadcast with $(\texttt{COIN}, v, p)$ at some time t. Now, consider three groups of nodes:

- For any node i that was in $\mathcal{A}_p$ before time t: i completes MAC-FirstMover for phase p after receiving and processing m or j's follow-up broadcast.
- For any node i that has not executed MAC-FirstMover of phase p by time t: i would "jump" to phase p after receiving and processing m or j's follow-up broadcast.
- For any node i that has already completed MAC-FirstMover of phase p before time t: this is trivial. Note that this is possible if i processes message(s) faster than j does, or there is a coin broadcast other than m. ◁

E Proof of Claim 15

Proof of Claim 15. In the framework of [3], if every node that has not crashed obtains the same output from the conciliator object, then all the fault-free nodes are guaranteed to terminate in the next phase. This design, the definition of a successful coin broadcast, and the ability to jump to a higher phase in MAC-RBC2 imply the claim. This is because for all nodes that update its state v_i in phase p, they must use the same outcome from the conciliator object (the value field of the successful coin broadcast). For the other nodes that jump to phase $p+1$ (from a phase $< p$), they must either receive phase-p coin broadcast(s) or receive the messages from the adopt-commit object in phase $p+1$. These messages and phase-p coin broadcasts (both the one and only original coin broadcast and follow-up coin broadcasts) must contain exactly the same value. ◁

F Proof of Theorem 16

Proof. First, we can decompose the total number of broadcasts by all fault-free nodes, denoted by N, into three components $N = N^{RBC} + N^O + N^F$, where (i) N^{RBC} denotes the number of broadcasts required by the part of adopt-commit (i.e., all the communication in Algorithm 3); (ii) N^O denotes the number of original broadcasts used in MAC-FirstMover for all phases; and (iii) N^F denotes the number of follow-up broadcasts used in MAC-FirstMover for all phases.

Let P denote the random variable of the first phase index in which the agreement is achieved, i.e., all nodes that have not crashed begin with same v in this phase.

First observe that in each phase, each node makes $O(1)$ broadcasts for adopt-commit and one follow-up broadcast in for MAC-FirstMover. Therefore, $N^{RBC} + N^F = O(nP)$. The rest of the proof focuses on bounding N^O.

Let $n'_p = 2^{\lfloor p/c \rfloor} n_0$ denote the input to MAC-FirstMover, namely the estimated system size in phase p. Then $n'_p \geq n$ for all $p \geq c(1 + \log_2(n/n_0))$. Therefore, Lemma 14 implies that the event E_p of no agreement in phase p has bounded probability $\mathbb{P}\{E_p\} \leq 1 - 0.05$ for all $p \geq c(1 + \log_2(n/n_0))$. Consequently,

$$\mathbb{P}\{P > c(1 + \log_2(n/n_0)) + q\} \leq \prod_{i=1}^{q} \mathbb{P}\{E_{\lfloor c(1+\log_2(n/n_0)) \rfloor + i}\} \leq \prod_{i=1}^{q} (1 - 0.05) \leq \exp(-0.05q).$$

Consequently, let $p^* = c(1 + \log_2(n/n_0)) + \frac{\ln(2/\delta)}{0.05}$. Upon substituting $q = \frac{\ln(2/\delta)}{0.05}$ into the previous bound, we have

$$\mathbb{P}\{P > p^*\} \leq \delta/2. \tag{4}$$

Note that with $c = \frac{\ln(2/\delta)}{0.05}$, $p^* = \frac{\ln(2/\delta)}{0.05}(2 + \log_2(n/n_0)) = O(\ln(n)\ln(1/\delta))$.

Let N_p^O denote the number of original broadcasts made byfault-free nodes in MAC-FirstMover of phase p. Lemma 12 implies that with probability $\geq 1 - \delta$, $N_p^O \leq 2n'_p \ln(1/\delta)$. Therefore, upon applying union bound, we have with probability $\geq 1 - \delta/2$,

$$\begin{aligned}
\sum_{p=1}^{p^*} N_p^O &\leq \sum_{p=1}^{p^*} 2n'_p \ln(2p^*/\delta) && \text{(recall that } n'_p = 2^{\lfloor p/c \rfloor} n_0\text{)} \\
&\leq 2cn_0 \ln(2p^*/\delta) \sum_{q=1}^{\lceil p^*/c \rceil} 2^p \\
&\leq 4cn_0 \ln(2p^*/\delta) 2^{\lceil p^*/c \rceil} && \text{(substitute definition of } p^*\text{)} \\
&= 4n_0 \frac{\ln(2/\delta)}{0.05} \ln\left(\frac{2\ln(2/\delta)(2 + \log_2(n/n_0))}{0.05\delta}\right) \exp_2\left(2 + \log_2(n/n_0)\right) \\
&= 320n \ln(2/\delta) \ln\left(\frac{2\ln(2/\delta)(2 + \log_2(n/n_0))}{0.05\delta}\right) \\
&= O\left(n \ln(1/\delta) \ln\left(\frac{\ln(n)\ln(1/\delta)}{\delta}\right)\right).
\end{aligned}$$

Equivalently, we have

$$\mathbb{P}\left\{\sum_{p=1}^{p^*} N_p^O > 320n \ln(2/\delta) \ln\left(\frac{2\ln(2/\delta)(2 + \log_2(n/n_0))}{0.05\delta}\right)\right\} \leq \delta/2. \tag{5}$$

Upon combining Equations (4), (5) and applying union bound, we have with probability $\geq 1 - \delta$, MAC-RBC2 achieves agreement (and thus termination) with

$$N = N^{RBC} + N^O + N^F = O\left(n \ln(n) \ln(1/\delta)\right)$$ ◀

Brief Announcement: Distributed Maximum Flow in Planar Graphs

Yaseen Abd-Elhaleem
Department of Computer Science, University of Haifa, Israel

Michal Dory
Department of Computer Science, University of Haifa, Israel

Merav Parter
Department of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot, Israel

Oren Weimann
Department of Computer Science, University of Haifa, Israel

Abstract

The dual of a planar graph G is a planar graph G^* that has a vertex for each face of G and an edge for each pair of adjacent faces of G. The profound relationship between a planar graph and its dual has been the algorithmic basis for solving numerous (centralized) classical problems on planar graphs involving distances, flows, and cuts. In the distributed setting however, the only use of planar duality is for finding a recursive decomposition of G [DISC 2017, STOC 2019].

In this paper, we extend the distributed algorithmic toolkit (such as recursive decompositions and minor-aggregations) to work on the dual graph G^*. These tools can then facilitate various algorithms on G by solving a suitable dual problem on G^*. Given a directed planar graph G with hop-diameter D, our key result is an $\tilde{O}(D^2)$-round algorithm[1] for Single Source Shortest Paths on G^*, which then implies an $\tilde{O}(D^2)$-round algorithm for Maximum *st*-Flow on G. Prior to our work, no $\tilde{O}(\text{Poly}(D))$-round algorithm was known for Maximum *st*-Flow. We further obtain a $D \cdot n^{o(1)}$-rounds $(1+\epsilon)$-approximation algorithm for Maximum *st*-Flow on G when G is undirected and s and t lie on the same face. Finally, we give a near optimal $\tilde{O}(D)$-round algorithm for computing the weighted girth of G.

The main challenges in our work are that G^* is not the communication graph (e.g., a vertex of G is mapped to multiple vertices of G^*), and that the diameter of G^* can be much larger than D (i.e., possibly by a linear factor). We overcome these challenges by carefully defining and maintaining subgraphs of the dual graph G^* while applying the recursive decomposition on the primal graph G. The main technical difficulty, is that along the recursive decomposition, a face of G gets shattered into (disconnected) components yet we still need to treat it as a dual node.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Maximum flow, shortest paths, planar graphs, distributed computing

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.40

Funding O. Weimann was supported in part by Israel Science Foundation grant 810/21.

1 Introduction

Distributed algorithms for network optimization problems have a long and rich history. These problems are commonly studied under the CONGEST model [18] where the network is abstracted as an n-vertex graph $G=(V,E)$ with hop-diameter D; communications occur in synchronous rounds, and per round, $O(\log n)$ bits can be sent along each edge. A sequence of

[1] The $\tilde{O}(\cdot)$ notation is used to omit poly $\log n$ factors.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 40; pp. 40:1–40:8
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

breakthrough results provided $\tilde{O}(D+\sqrt{n})$-round algorithms for fundamental graph problems, such as minimum spanning tree (MST) [6], approximate shortest-paths [16], minimum cuts [3], and approximate flow [9]. For general n-vertex graphs $\tilde{O}(D+\sqrt{n})$ rounds is known to be near optimal, existentially [20].

A major and concentrated effort has been invested in designing improved solutions for special graph families that escape the topology of the worst-case lower bound graphs of [20]. The lower bound graph is sparse, and of arboricity two, so it belongs to many graph families. Arguably, one of the most interesting non-trivial families that escapes it, is the family of planar graphs. Thus, a significant focus has been given to the family of planar graphs, due to their frequent appearance in practice and because of their rich structural properties. In their seminal work, Ghaffari and Haeupler [7, 8] initiated the line of distributed planar graph algorithms based on the notion of *low-congestion shortcuts*. The latter serves the communication backbone for obtaining $\tilde{O}(D)$-round algorithms for MST [8], minimum cut [8, 11] and approximate shortest paths [21, 22] in planar graphs.

An additional key tool in working with planar graphs, starting with the seminal work of Lipton and Tarjan [15], is that of a planar *separator path*: a path whose removal from the graph leaves connected components that are a constant factor smaller. Ghaffari and Parter [10] presented a $\tilde{O}(D)$-round randomized algorithm for computing a cycle separator of size $O(D)$ which consists of a separator path plus one additional edge (that is possibly a *virtual* edge that is not in G). By now, planar separators are a key ingredient in a collection of $\tilde{O}(\text{poly}(D))$-round solutions for problems such as DFS [10], distance computation [14], and reachability [17]. An important aspect of the planar separator algorithm of [10] is that it employs a computation on the dual graph, by communicating over the primal graph.

Primal maximum flow via dual SSSP. Our goal in this paper is to expand the algorithmic toolkit for performing computation on the dual graph. This allows us to exploit the profound algorithmic duality in planar graphs, in which solving a problem A in the dual graph provides a solution for problem B in the primal graph. Within this context, our focus is on the *Maximum st-Flow* problem (in directed planar graphs with edge capacities) which asks to compute the maximum amount of flow that can be sent from a source vertex s to a target vertex t while respecting edge capacities. The Maximum st-flow problem is arguably one of the most classical problems in theoretical computer science, extensively studied since the 50's, and still admitting breakthrough results in the sequential setting, such as the recent almost linear time algorithm by Chen, Kyng, Liu, Peng, Gutenberg and Sachdeva [1]. Despite persistent attempts over the years, our understanding of the distributed complexity of the Maximum st-flow problem is still quite lacking. For general *undirected* n-vertex graphs, there is a $(1+o(1))$-approximation algorithm that runs in $(\sqrt{n}+D)n^{o(1)}$ rounds, by Ghaffari, Karrenbauer, Kuhn, Lenzen and Patt-Shamir [9]. For directed n-vertex *planar* graphs, a $D \cdot n^{1/2+o(1)}$-round exact algorithm has been given by de Vos [2]. No better tradeoffs are known for undirected planar graphs. In lack of any $\tilde{O}(\text{poly}(D))$-round maximum st-flow algorithm for directed planar graphs (not even when allowing approximation) we ask:

▶ **Question 1.1.** *Is it possible to compute the maximum st-flow in directed planar graphs within $\tilde{O}(poly(D))$ rounds?*

In directed planar graphs with integral edge-capacities, it is known from the 80's [23] that the maximum st-flow can be found by solving at most $\log \lambda$ instances of *Single Source Shortest Paths* (SSSP) with positive and negative edge-lengths on the *dual* graph G^*, where λ is the maximum st-flow value. Their algorithm exploits the fact that any capacity-respecting flow in G can be decomposed into (1) a not necessarily capacity-respecting st-flow of the same value, and (2) a feasible *circulation*. Since G is planar, a feasible circulation can be obtained by a *feasible potential* over its faces (i.e., nodes of G^*); It is known that distances in G^* from any source constitute a feasible potential over its nodes. Hence, dual SSSP immediately implies primal maximum st-flow. We answer Question 1.1 in the affirmative by designing a $\tilde{O}(D^2)$-round SSSP algorithm on the dual graph G^*. Our algorithm works in the most general setting (i.e. when G^* is directed and has positive and negative integral edge-lengths) and matches the fastest known exact SSSP algorithm in the primal graph. We show:

▶ **Theorem 1.2** (Exact Maximum st-Flow in Directed Planar Graph)**.** *There is a randomized distributed algorithm that given an n-vertex directed planar communication network G with hop-diameter D and integral edge-capacities, and two vertices s, t, computes the maximum st-flow value and assignment in $\tilde{O}(D^2)$ rounds.*

No prior $\tilde{O}(\text{poly}(D))$ algorithm has been known for this problem, not even when allowing a constant approximation. We further improve the running time to $D \cdot n^{o(1)}$ rounds for the case of a $(1+\epsilon)$-approximation, provided that G is undirected and that s and t both lie on the same face:

▶ **Theorem 1.3** (Approximate Maximum st-Flow in Undirected st-Planar Graphs)**.** *There is a randomized distributed algorithm that given an n-vertex undirected planar communication network G with hop-diameter D and integral edge-capacities, and two vertices s, t lying on the same face, computes a $(1+\epsilon)$-approximation of the maximum st-flow value and a matching assignment in $D \cdot n^{o(1)}$ rounds.*

This latter result is also obtained by exploiting the duality between flows and distances. Our algorithm is based on an approximate SSSP algorithm that runs in $D \cdot n^{o(1)}$ rounds in planar graphs [22]. Our implementation of the algorithm on the dual graph matches its round complexity in the primal graph. Our almost-optimal round complexity improves significantly over the current algorithm for general graphs that runs in $(\sqrt{n} + D)n^{o(1)}$ rounds [9].

Primal weighted girth via dual cuts. A distance parameter of considerable interest is the network *girth*. For unweighted graphs, the girth is the length of the smallest cycle in the graph. For weighted graphs, the girth is the cycle of minimal total edge weight. Distributed girth computation has been studied over the years mainly for general n-vertex unweighted graphs. Frischknecht, Holzer and Wattenhofer [5] provided an $\Omega(\sqrt{n})$-round lower bound for computing a $(2-\epsilon)$ approximation of the unweighted girth. The state-of-the-art upper bound for the unweighted girth problem is a $(2-\epsilon)$ approximation in $\tilde{O}(n^{2/3} + D)$ rounds, obtained by combining the works of Peleg, Roditty and Tal [19] and Holzer and Wattenhofer [12]. The weighted girth problem has been shown to admit a near-optimal lower bound of $\tilde{\Omega}(n)$ rounds in general graphs by Hua, Qian, Yu, Shi and Jin [13]. Turning to planar graphs, Parter [17] devised a $\tilde{O}(D^2)$ round algorithm for computing the weighted girth in directed planar graphs via SSSP computations. For undirected and unweighted planar graphs, the (unweighted) girth can be computed in $\tilde{O}(D)$ rounds by replacing the $\tilde{O}(D^2)$-round SSSP algorithm by a $O(D)$-round BFS algorithm. In light of this gap, we ask:

▶ **Question 1.4.** *Is it possible to compute the weighted girth of an undirected weighted planar graph within (near-optimal) $\tilde{O}(D)$ rounds?*

We answer this question in the affirmative by taking a different, non distance-related, approach than that taken in prior work. Our $\tilde{O}(D)$ round algorithm exploits the useful duality between cuts and cycles. We present a dual framework of the *minor-aggregation* model. Using it, we can simulate the primal exact minimum cut algorithm of Ghaffari and Zuzic [11] on the dual graph. This dual simulation matches the primal round complexity. The solution to the dual cut problem immediately yields a solution to the primal weighted girth problem. We show:

▶ **Theorem 1.5** (Planar Weighted Girth). *There is a randomized distributed algorithm that given an n-vertex undirected weighted planar communication network G with hop-diameter D, computes the weighted girth (and finds a corresponding cycle) in $\tilde{O}(D)$ rounds.*

As the algorithmic power of the minor-aggregation model is currently limited to undirected graphs, it will be interesting to devise improved girth algorithms for directed planar graphs as well.

2 Technical Overview

Our results are based on two main (primal) tools that we extend to work on the dual graph: Minor Aggregation and Bounded Diameter Decomposition. We highlight the key ideas of these techniques and the challenges encountered in their dual implementation. For all the algorithms that we implement in the dual graph, we match the primal round complexity.

2.1 Minor-Aggregations in the Dual

An important recent development in the field of distributed computing was a new model of computation, called the *minor-aggregation model* introduced by Zuzic ⓡ[2] Goranci ⓡ Ye ⓡ Haeupler ⓡ Sun [22], then extended by Ghaffari and Zuzic [11] to support working with *virtual nodes* added to the input graph. Recent state-of-art algorithms for various classical problems can be formulated in the minor-aggregation model (e.g., the exact min-cut algorithm of [11], and the undirected shortest paths approximation algorithms of [21, 22]). Motivated by the algorithmic power of this model, we provide an implementation of the minor aggregation model in the dual graph. The round complexity of our implementation matches its primal complexity. As noted by [22], minor aggregations can be implemented by solving the (simpler) part-wise aggregation task, where one needs to compute an aggregate function in a collection of vertex-disjoint connected parts of the graph. The planar separator algorithm of [10] implicitly implements a part-wise aggregation algorithm in the dual graph. Our contribution is in providing an explicit and generalized implementation of the dual part-wise aggregation problem and using it to implement the minor-aggregation model in the dual graph. We then use this algorithm for computing the exact minimum weighted cut in the dual graph, which by duality provides a solution to the weighted girth problem in the primal graph. We also use it to simulate the recent approximate SSSP by [22] in the dual graph, leading to our approximate max st-flow algorithm. Since currently there are fast SSSP minor-aggregation algorithms only for undirected graphs with positive weights, this approach leads to an approximate max st-flow algorithm in undirected planar graphs when s and t are on the same face. To solve the more general version of the max flow problem, we need additional tools described next.

[2] ⓡ is used to denote that the authors' ordering is randomized, as the authors ask to cite their work this way.

2.2 SSSP in the Dual

Bounded diameter decompositions. The Bounded Diameter Decomposition (BDD), introduced by Li and Parter [14] is an algorithmic tool for solving graph problems in a divide-and-conquer manner, in the CONGEST model. Intuitively, the BDD plays an analogous role to planar *separator decomposition* in the centralized setting, in the following sense. The centralized divide-and-conquer approach repetitively removes the separator vertices from the graph and recurses on the remaining subgraphs that are (a constant factor) smaller in size. For the algorithmic applications it is only important that the size of the separator and the remaining subgraphs are small. In the distributed setting, it is desired to obtain a separator of $O(D)$ size in all recursive subgraphs, allowing a fast ($\tilde{O}(\text{poly}(D))$-round) broadcast of separator related information (e.g. pairwise distances), which is in particular useful for a divide-and-conquer approach. While this can be obtained in the first recursion level, once we remove the first separator, the remaining subgraphs are smaller in size, but they may have considerably larger diameter, even up to $\Theta(n)$. Allowing the algorithm to use the other subgraphs, to provide shortcut paths, creates the possibility of *congestion* as now many subproblems may need to use the same edge. These two opposing dilation and congestion forces are settled by the BDD algorithm, in a near optimal manner. The BDD provides a hierarchical graph decomposition of $O(\log n)$ layers. The subgraphs (called bags) obtained in each recursive level are nearly edge-disjoint (sharing only the edges of the separator) and of diameter $\tilde{O}(D)$. I.e., allowing one to apply an algorithm on all bags of the same level simultaneously without incurring more than a $\text{poly}\log n$ factor overhead in the round complexity of running the same algorithm on the original network of communication (which has a small diameter of D). There might be as many as $\tilde{O}(D)$ children of a bag (all a constant factor smaller than their parent bag). However, the number of child bags has no importance, as we can work on all of them in parallel. BDDs have proven to be useful for divide-and-conquer CONGEST algorithms on the (primal) graph G (e.g. distance labeling, diameter approximation, routing schemes and reachability [4, 14, 17]).

Our approach: recurse on primal, solve on dual. Due to the wide applicability of BDDs for solving graph problems in planar graphs, we would like to exploit them also for solving problems on the dual graph. A natural approach could be to simulate a BDD algorithm on the dual network. However, there are several barriers. First, it is unclear how to simulate a general algorithm on the dual network, as this is not our communication network. Second, the diameter of the dual graph can be large (possibly linear) and the running time of the algorithm depends on the graph diameter. To overcome it we take a different approach. We apply a divide-and-conquer approach on the dual graph G^* by using the BDD computation on the primal graph G. Taking a dual lens on the primal BDD introduces several challenges that arise when one needs to define the dual bags from the given primal bags. This primal to dual translation is rather non-trivial due to critical gaps that arise when one needs to maintain information w.r.t faces of G, rather than vertices of G, over the recursive BDD procedure, as we elaborate next.

Challenge I: shattered faces. Throughout, we refer to faces of the primal graph G as *nodes* (rather than vertices) of the dual graph G^*. In the primal graph, a vertex is an atomic unit, which keeps its identity throughout the computation. The situation in the dual graph is considerably more involved. Consider a constant diameter (primal) graph with a face f with $\Theta(n)$ edges. Throughout the recursive BDD, the vertices of the face f are split among multiple faces, and eventually f is shattered among possibly a linear number of leaf bags.

This means that a node in a dual bag does no longer correspond to a face f of the primal bag, but rather to a subset of edges of f. This creates a challenge in the divide-and-conquer computation, where one needs to assemble fragments of information from multiple bags.

Challenge II: virtual edges. For the BDD implementation, it is crucial for the separator to be a *simple cycle*. This was obtained in [10] by adding a single artificial (virtual) edge. The virtual edges are embedded in a way that preserve planarity, but they require special treatment since they are not part of the communication graph. In the primal BDD, the role of the virtual edge is limited to defining the child bags, and can be discarded afterwards. This use-and-forget mindset can no longer be applied in our setting. We elaborate. In a primal divide-and-conquer algorithm, the separator is thought of as a subset of vertices where each path in G from one side of the separator to the other side must intersect the separator at a *vertex*. In our case, since we are working with the dual graph, we have that paths in G^* from one side of separator to the other side must intersect the separator at an *edge*. That is, we view the separator as an edge-separator (i.e., a cut in the dual graph) not a vertex-separator. This is challenging because now we need to take into account the virtual edge that is not a real edge of G but is an edge of the separator.

Our approach. To deal with the above challenges, we work as follows. First, we analyze the way faces are partitioned during the BDD algorithm. We prove that in each bag X of the BDD there is at most one face of G that can be partitioned between the different child bags of X and was not partitioned in previous levels, this is exactly the face f that contains the virtual edge of the bag. We call the different parts of f that appear in different child bags *face-parts*. Since the decomposition has $O(\log n)$ levels, overall we have at most $O(\log n)$ face-parts in each bag. For a bag X, we define a dual bag X^* as follows. The nodes of X^* are the faces and face-parts of G that appear in X, where two nodes are connected by a dual edge if they share a primal edge in X. If $X = G$, this is exactly the dual graph G^*. The nodes g in X^* will be simulated by the vertices of the corresponding face or face-part, and each dual edge adjacent to g will be known by one of these vertices.

Our next goal is to use the decomposition in order to compute distances in the dual graph. More concretely, we compute distance labels. Each node in a bag X^* gets a short label of size $\tilde{O}(D)$, such that given the labels of two nodes in X^* we can deduce their distance. We take a recursive approach. We first compute distance labels in the child bags of a dual bag X^* and then combine them to compute distances in X^*. To do so, we identify a set of $\tilde{O}(D)$ special nodes F_X in X^*, that contain nodes adjacent to the separator, as well as nodes corresponding to faces or face-parts that are partitioned between child bags of X^*. We prove that any shortest path in X^* is either entirely contained in one of the child bags (and hence we already computed the distances recursively), or has a special node in F_X. Hence, it is enough to store in the label of a node g its distances from nodes in F_X and its label in the child bag of X^* that contains g (if g is partitioned to several child bags, g is in F_X, and in this case we just store the distances to nodes in F_X without a recursive label). Finally, we broadcast $\tilde{O}(D^2)$ information that includes labels of nodes in F_X (or corresponding face-parts) in the child bags, and the edges of the separator, and prove that based on this information nodes can deduce locally their distance label in X^*. This follows as each shortest path is either entirely contained in a child bag, or can be broken up to subpaths whose endpoints are in F_X, and are either entirely contained in a child bag (and hence their distance can be deduced from the labels we broadcast), or use a separator edge between different child bags (we broadcast all these edges), or use face-parts of the same face that are contained in different child bags (in this case, we connect the corresponding face-parts with a zero weight edge).

References

1 Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *63rd FOCS*, pages 612–623, 2022. doi:10.1109/FOCS54457.2022.00064.

2 Tijn de Vos. Minimum cost flow in the congest model. In *30th SIROCCO*, pages 406–426, 2023. doi:10.1007/978-3-031-32733-9_18.

3 Michal Dory, Yuval Efron, Sagnik Mukhopadhyay, and Danupon Nanongkai. Distributed weighted min-cut in nearly-optimal time. In *53rd STOC*, pages 1144–1153, 2021. doi:10.1145/3406325.3451020.

4 Jinfeng Dou, Thorsten Götte, Henning Hillebrandt, Christian Scheideler, and Julian Werthmann. Brief announcement: Distributed construction of near-optimal compact routing schemes for planar graphs. In *41tst PODC*, pages 67–70, 2023. doi:10.1145/3583668.3594561.

5 Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *23rd SODA*, pages 1150–1162, 2012. doi:10.1137/1.9781611973099.91.

6 Juan A. Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. Comput.*, 27(1):302–316, 1998. doi:10.1137/S0097539794261118.

7 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks I: planar embedding. In *34th PODC*, pages 29–38, 2016. doi:10.1145/2933057.2933109.

8 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, mst, and min-cut. In *27th SODA*, pages 202–219, 2016. doi:10.1137/1.9781611974331.CH16.

9 Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. Near-optimal distributed maximum flow. In *33rd PODC*, pages 81–90, 2015.

10 Mohsen Ghaffari and Merav Parter. Near-Optimal Distributed DFS in Planar Graphs. In *31st DISC*, pages 21:1–21:16, 2017. doi:10.4230/LIPICS.DISC.2017.21.

11 Mohsen Ghaffari and Goran Zuzic. Universally-optimal distributed exact min-cut. In *40th PODC*, pages 281–291, 2022. doi:10.1145/3519270.3538429.

12 Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *30th PODC*, pages 355–364, 2012. doi:10.1145/2332432.2332504.

13 Qiang-Sheng Hua, Lixiang Qian, Dongxiao Yu, Xuanhua Shi, and Hai Jin. A nearly optimal distributed algorithm for computing the weighted girth. *Sci. China Inf. Sci.*, 64(11), 2021. doi:10.1007/S11432-020-2931-X.

14 Jason Li and Merav Parter. Planar diameter via metric compression. In *51st STOC*, pages 152–163, 2019. doi:10.1145/3313276.3316358.

15 Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

16 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *46th STOC*, pages 565–573, 2014. doi:10.1145/2591796.2591850.

17 Merav Parter. Distributed planar reachability in nearly optimal time. In *34th DISC*, volume 179, pages 38:1–38:17, 2020. doi:10.4230/LIPICS.DISC.2020.38.

18 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000.

19 David Peleg, Liam Roditty, and Elad Tal. Distributed algorithms for network diameter and girth. In *39th ICALP*, pages 660–672, 2012. doi:10.1007/978-3-642-31585-5_58.

20 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In *43rd STOC*, pages 363–372, 2011. doi:10.1145/1993636.1993686.

21 Václav Rozhon ⓡ Christoph Grunau ⓡ Bernhard Haeupler ⓡ Goran Zuzic ⓡ Jason Li. Undirected (1+ϵ)-shortest paths via minor-aggregates: near-optimal deterministic parallel and distributed algorithms. In *54th STOC*, pages 478–487, 2022.

22 Goran Zuzic ⓡ Gramoz Goranci ⓡ Mingquan Ye ⓡ Bernhard Haeupler ⓡ Xiaorui Sun. Universally-optimal distributed shortest paths and transshipment via graph-based l_1-oblivious routing. In *33rd SODA*, pages 2549–2579, 2022.

23 Shankar M. Venkatesan. *Algorithms for network flows.* Ph.D. thesis, Pennsylvania State University, 1983.

Brief Announcement: Towards Optimal Communication Byzantine Reliable Broadcast Under a Message Adversary

Timothé Albouy
Univ Rennes, Inria, CNRS, IRISA,
35042 Rennes-cedex, France

Davide Frey
Univ Rennes, Inria, CNRS, IRISA,
35042 Rennes-cedex, France

Ran Gelles
Bar-Ilan University, Ramat Gan, Israel

Carmit Hazay
Bar-Ilan University, Ramat Gan, Israel

Michel Raynal
Univ Rennes, Inria, CNRS, IRISA,
35042 Rennes-cedex, France

Elad Michael Schiller
Chalmers University of Technology,
Gothenburg, Sweden

François Taïani
Univ Rennes, Inria, CNRS, IRISA,
35042 Rennes-cedex, France

Vassilis Zikas
Purdue University, West Lafayette, IN, USA

Abstract

We address the problem of Reliable Broadcast in asynchronous message-passing systems with n nodes, of which up to t are malicious (faulty), in addition to a *message adversary* that can drop some of the messages sent by correct (non-faulty) nodes. We present a Message-Adversary-Tolerant Byzantine Reliable Broadcast (MBRB) algorithm that communicates an almost optimal amount of $\mathcal{O}(|m| + n^2\kappa)$ bits per node, where $|m|$ represents the length of the application message and $\kappa = \Omega(\log n)$ is a security parameter. This improves upon the state-of-the-art MBRB solution (Albouy, Frey, Raynal, and Taïani, TCS 2023), which incurs communication of $\mathcal{O}(n|m| + n^2\kappa)$ bits per node. Our solution sends at most $4n^2$ messages overall, which is asymptotically optimal. Reduced communication is achieved by employing coding techniques that replace the need for all nodes to (re-)broadcast the entire application message m. Instead, nodes forward authenticated fragments of the encoding of m using an erasure-correcting code. Under the cryptographic assumptions of PKI and collision-resistant hash, and assuming $n > 3t+2d$, where the adversary drops at most d messages per broadcast, our algorithm allows at least $\ell = n - t - (1+\epsilon)d$ (for any $\epsilon > 0$) correct nodes to reconstruct m, despite missing fragments caused by the malicious nodes and the message adversary.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Asynchronous message-passing, Byzantine fault-tolerance, Message adversary, Reliable Broadcast

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.41

Related Version *Full Version*: `https://arxiv.org/abs/2312.16253` [1]

Funding This work was partially supported by the French ANR projects ByBloS (ANR-20-CE25-0002-01) and PriCLeSS (ANR-10-LABX-07-81), devoted to the modular design of building blocks for large-scale Byzantine-tolerant multi-users applications. Research supported in part by the United States-Israel Binational Science Foundation (BSF) through Grant No. 2020277.

Acknowledgements R. Gelles would like to thank Paderborn University and CISPA – Helmholtz Center for Information Security for hosting him while part of this research was done.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 41; pp. 41:1–41:7
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Byzantine Reliable Broadcast (*BRB* for short) allows n asynchronous nodes to agree eventually on a message sent by a designated node, the *sender*, despite the possible malicious (Byzantine) behavior by some nodes and the transmission network [4]. Byzantine reliable broadcast plays a crucial role in several key applications, including consensus algorithms, replication, event notification, and distributed file systems, among others. These systems sometimes require broadcasting large messages or files (e.g., permissioned blockchains), and thus, reducing the communication overhead to a minimum is an important aspect of achieving scalability. In that vein, this work aims at providing *communication efficient* solutions for the task of reliable broadcast in the presence of node and link faults.

A significant challenge to reliable broadcast algorithms arises when the message-passing system is unreliable and possibly cooperates with the Byzantine nodes. Link faults [11, 12] give Byzantine nodes (potentially limited) control over certain network links, enabling them to omit or corrupt messages (an ability captured under the umbrella term *message adversary* [9]). This work focuses on a specific type of *message adversary* [9] that can only omit messages sent by correct nodes, but that cannot alter their content. This message adversary abstracts cases related to *silent churn*, where nodes may voluntarily or involuntarily disconnect from the network without explicitly notifying other nodes.

Problem overview. We assume n nodes over an asynchronous network, where a message can be delayed for an arbitrary yet finite amount of time (unless omitted by the message adversary). We assume the existence of t Byzantine nodes and a message adversary capable of omitting up to d messages per node's broadcast. To be more precise, a node communicates through a comm primitive (or a similar multicast/unicast primitive that targets a dynamically defined subset of processes), which results in the transmission of n messages, with each node being sent one message, including the sender. The message adversary can choose to omit messages in transit to a subset of at most d correct processes. The adversary is only limited by the size of that subset. For instance, between different comm invocations, the adversary has the freedom to modify the set of correct processes to which messages are omitted. Furthermore, a designated sender node holds a message m that it wishes to broadcast to all the nodes.

An algorithm that satisfies the requirements of reliable broadcast despite Byzantine nodes and a message adversary is called a *Message-adversary Byzantine Reliable Broadcast* (MBRB) algorithm. The detailed version of MBRB's requirements was formulated in [2], see Section 2.

Background. Albouy, Frey, Raynal, and Taïani [2] recently proposed a Message-adversary Byzantine Reliable Broadcast algorithm (which we denote AFRT for short) for asynchronous networks that withstands the presence of t Byzantine nodes and a message adversary capable of omitting up to d messages per node's broadcast. AFRT guarantees the reliable delivery of any message when $n >$ 3t+2d. Moreover, they demonstrate the necessity of this bound on the number of Byzantine nodes and omitted messages, as no reliable broadcast algorithm exists otherwise.

One caveat of AFRT regards its communication efficiency. While it achieves an optimal number of $\mathcal{O}(n^2)$ messages, and an optimal delivery power $\ell = n - t - d$, each node's communication requires $\mathcal{O}(n \cdot (|m| + n\kappa))$ bits, where $|m|$ represents the number of bits in the broadcast message and κ is the length of the digital signatures used in their algorithm. In the current work, we design an algorithm that significantly reduces the communication cost per node while preserving the total number of messages communicated. Our solution features at most $4n$ messages per correct node (corresponding to $4n^2$ messages overall), and only $\mathcal{O}(|m| + n^2\kappa)$ bits per correct node. Overall, $\mathcal{O}(n|m| + n^3\kappa)$ bits are communicated by correct nodes. Reducing the second term to $(n^2 \log n)\kappa$ can be done by employing standard techniques of *threshold signatures*, which replace the need to communicate a quorum of signatures; see, e.g., [3]. Note that $\Omega(n|m| + n^2\kappa)$ is a straightforward lower bound on the overall communication for deterministic algorithms using signatures (up to the size of the signature), see [5, 8], as every correct node must receive the message m, and as the reliable broadcast of a single bit necessitates at least $\Omega(n^2)$ messages [6].

Contributions. This work is the first to present an MBRB algorithm tolerating an hybrid adversary combining t Byzantine nodes and a Message Adversary of power d, while providing optimal Byzantine resilience, near-optimal communication, and near-optimal delivery power ℓ.

2 Preliminaries

General notations and conventions. For a positive integer n, let $[n]$ denote the set $\{1, 2, \ldots, n\}$. A sequence of elements $(x_1, \ldots, x_n)$ is shorthanded as $(x_i)_{i\in[n]}$. We use the symbol '-' to indicate any possible value. That is, $(h, \text{-})$ means a tuple where the second index includes any arbitrary value which we do not care about. All logarithms are base 2.

Nodes and Network. We focus on asynchronous message-passing systems that have no guarantees of communication delay. Also, the algorithm cannot explicitly access the clock or use timeouts. The system consists of a set, $\mathcal{P} = \{p_1, \ldots, p_n\}$, of n fail-prone nodes (or processes). We identify party i with p_i.

Communication means. Any ordered pair of nodes $p_i, p_j \in \mathcal{P}$ has access to a communication channel, $channel_{i,j}$. Each node can send messages to all nodes (possibly by sending a different message to each node). That is, any node, $p_i \in \mathcal{P}$, can invoke the transmission macro, $\mathsf{comm}(m_1, \ldots, m_n)$, that communicates the message m_j to p_j over $channel_{i,j}$. The message m_j can also be empty, in which case nothing will be sent to p_j. However, in our algorithms, all messages sent in a single comm activation will have the same length. Furthermore, when a node sends the same message m to all nodes, we write $\mathsf{broadcast}(m) = \mathsf{comm}(m, m, \ldots, m)$ for shorthand. We call each message m_j transmitted by the protocol an *implementation message* (or simply, a *message*) to distinguish such messages from the *application*-level messages, i.e., the one the sender wishes to broadcast.

Byzantine nodes. Faulty nodes are called *Byzantine* and their adversarial behavior can deviate from the proposed algorithm in any manner. They might perform any arbitrary computation, and we assume their computing power is at least as strong as that of non-faulty nodes, yet not as strong as to undermine the security of the cryptographic signatures we use (see below). We assume that, at most, t nodes are faulty, where t is a value known to the nodes. Non-faulty nodes are called *correct nodes.* The set of correct nodes contains c nodes where $n - t \leq c \leq n$. The value of c is unknown.

Message adversary. This entity can remove implementation messages from the communication channels used by correct nodes when they invoke $\mathsf{comm}(\cdot)$. More precisely, during each $\mathsf{comm}(m_1, \ldots, m_n)$ call, the adversary has the discretion to eliminate up to d messages in the set $\{m_i\}$ from their corresponding communication channels where they were queued. Similar to [2], we assume $n > 3t{+}2d$.

Error Correction Codes. A central tool used in our algorithm is an error-correction code (ECC) [10]. Intuitively speaking, an ECC takes a message as input and adds redundancy to create a codeword from which the original message can be recovered even when parts of the codeword are corrupted. In this work, we focus on *erasures*, a corruption that replaces a symbol of the codeword with a special erasure mark $\perp$.

Cryptographic Primitives. Our algorithm relies on cryptographic assumptions. We assume that the Byzantine nodes are computationally bounded with respect to the security parameter, denoted by κ. That is, all cryptographic algorithms are polynomially bounded in the input 1^κ. We assume that $\kappa = \Omega(\log n)$.

Hash functions. A collision-resistant hash is a function[1] $\mathsf{hash} : \{0,1\}^* \to \{0,1\}^\kappa$ that satisfies the following *collision resistance* property: For any computationally bounded algorithm A and any $x \in \{0,1\}^*$, $\Pr[A(x) = x' \wedge \mathsf{hash}(x') = \mathsf{hash}(x)] < 2^{-\Omega(\kappa)}$. I.e., finding a pair x, x' with the same hash is infeasible, except with negligible probability in the security parameter.

Signature schemes. A *digital signature* scheme is a pair of possibly randomized algorithms $\mathsf{SIG} = (\mathsf{sign}, \mathsf{Verify})$. The signing algorithm executed by node p_i (denoted, sign_i) takes a message m and implicitly a private key. It then produces a signature $\sigma = \mathsf{sign}_i(m)$. The verifying algorithm takes a message, its corresponding signature, and the identity of the signer (and implicitly a public key), and outputs a single bit, $b = \mathsf{Verify}(m, \sigma, i)$, which indicates whether the signature is valid or not, $b \in \{\mathsf{valid}, \mathsf{invalid}\}$.

Merkle Trees [7]. These are means to commit to a message composed of several fragments so that one can prove, for each fragment independently, that it belongs to the committed message. This primitive is parameterized by a security parameter κ and consists of two functions: $\mathsf{MerkleTree}(\cdot)$ which generates the proofs for each fragment of the message, and $\mathsf{VerifyMerkle}(\cdot)$, which given a fragment along with its proof, verifies that the fragment indeed belongs to the committed message.

Specification of the MBRB primitive. The Objective of MBRB is to guarantee a reliable delivery of a message while upholding specific safety and liveness criteria, despite actions taken by Byzantine nodes and the message adversary An MBRB algorithm contains the MBRB-broadcast and MBRB-deliver operations.

Definition 1 specifies the safety and liveness properties. Safety ensures that messages are delivered correctly without spurious messages, duplication, or duplicity. The liveness guarantee that if a correct node broadcasts a message, it will eventually be delivered by at least one correct node (MBRB-Local-delivery), and that if a correct node delivers a message from any specific sender, that message will eventually be delivered by a sufficient number, ℓ, of correct nodes (MBRB-Global-delivery), where ℓ is a measure of the *delivery power* of the MBRB algorithm and might depend on the adversary's power, i.e., on t and d.

▶ **Definition 1.** *An MBRB is an algorithm that satisfies the following properties.*

- ***MBRB-Validity.*** *Suppose p_s is correct and a correct node, p_i, MBRB-delivers an application message m. Then, node p_s has MBRB-broadcast m (before that MBRB-delivery).*
- ***MBRB-No-duplication.*** *A correct node p_i MBRB-delivers at most one application message m.*
- ***MBRB-No-duplicity.*** *No two different correct nodes MBRB-deliver different application messages from node p_s.*
- ***MBRB-Local-delivery.*** *Suppose p_s is correct and MBRB-broadcasts an application message m. At least one correct node, p_j, eventually MBRB-delivers m from node p_s.*
- ***MBRB-Global-delivery.*** *Suppose a correct node, p_i, MBRB-delivers an application message m from p_s. Then, at least ℓ correct nodes MBRB-deliver m from p_s.*

[1] Formally speaking, a hash function must be chosen randomly from a family of possible hash functions. Otherwise, an adversarial algorithm A exists. We avoid a formal treatment of this issue in our paper. In practice, a fixed function is used (e.g., SHA2 or SHA3).

3 The Coded-MBRB algorithm

The proposed solution, named Coded MBRB (Algorithm 2), allows a distinguished sender p_s to disseminate one specific application message m. In the description below, we assume there is a single sender, p_s, and all nodes know its identity p_s. In the full version [1], we discuss how to extend this algorithm so that it implements a general MBRB algorithm, allowing any node to be the sender, as well as allowing multiple instances of the MBRB, either with the same or different senders, to run concurrently.

Algorithm description. MBRB-BROADCAST(m) (line 6) allows the sender to start disseminating the application message, m. It is designed to be executed by the sender process, p_s. The initial step of the sender (line 7) invokes COMPUTEFRAGMERKLETREE(m) (Algorithm 1), which encodes the message m using an error-correction code, divides it into n fragments and constructs a Merkle tree that includes the different fragments. The function returns several essential values: the Merkle root hash h, and the fragment details $(\tilde{m}_j, \pi_j, j)$, which contains the fragment data itself $\tilde{m}_j$ (the j-th part of the codeword $\mathsf{ECC}(m)$), a proof of inclusion π_j for that part, and the respective index j of each fragment.

The sender node, p_s, is responsible for signing the computed Merkle root hash h and generating a signature, denoted sig_s (line 8). Notably, this signature includes p_s's identifier. The sender then initiates m's propagation by employing the operation COMM (line 9), which sends to each process p_j the Merkle root hash h, the j-th fragment details $(\tilde{m}_j, \pi_j, j)$, and the signature sig_s (line 8). When this message (or later messages communicated in the algorithm) is received by some node p_i, it first verifies that all the signatures and the Merkle proofs that the message contains are valid, and that p_s's signature is included in the messages; otherwise, the message is ignored. This action is encapsulated by ISVALID() (lines 11, 17, and 33).

The rest of the algorithm progresses in two phases. The first phase is responsible for message dissemination, which forwards message fragments received by the sender. The other role of this phase is reaching a quorum of nodes that vouch for the same message. A node vouches for a single message by signing its hash value. Nodes collect and store signatures until it is evident that sufficiently many nodes agree on the same message. The subsequent phase focuses on disseminating the quorum of signatures so that it is observed by at least ℓ correct nodes, and on successfully terminating while ensuring the delivery of the reconstructed message.

Algorithm 1 The COMPUTEFRAGMERKLETREE(m) function.

```
Function COMPUTEFRAGMERKLETREE($m$) is
    $\tilde{m} \leftarrow \mathsf{ECC}(m)$                ▷ Such that $m$ is recoverable from $k = \Omega(n)$ fragments
    let $\tilde{m}_1, \ldots, \tilde{m}_n$ be $n$ equal size fragments of $\tilde{m}$
    $(h, \pi_1, \ldots, \pi_n) \leftarrow \mathsf{MerkleTree}(\tilde{m}_1, \ldots, \tilde{m}_n)$ ;
    return $\big(h, (\tilde{m}_j, \pi_j, j)_{j \in [n]}\big)$
```

Analysis. The following theorem states that our algorithm is correct. Due to page limit, the complete proof and discussion on the assumptions appear in the full version [1].

▶ **Theorem 2** (Main). *Assume $n > 3t + 2d$, $k \leq (n - t - 2d)$ and $\varepsilon > 0$. Algorithm 2 implements an MBRB solution with $\ell > n - t - (1 + \varepsilon)d$. Any algorithm activation on the input message m communicates $4n^2$ messages, where each node communicates $O(|m| + n^2\kappa)$ bits overall.*

Algorithm 2 The Coded MBRB Algorithm (code for p_i, single-shot, single-sender).

```
6  Function MBRBbroadcast($m$) is                                  ▷ only executed by the sender, $p_s$
7      $\big(h, (\tilde{m}_j, \pi_j, j)_j\big) \leftarrow$ computeFragMerkleTree($m$)
8      $sig_s \leftarrow \big(\mathsf{sign}_s(h), s\big)$
9      comm($v_1, \ldots, v_n$) where $v_j = \langle \textsc{send}, h, (\tilde{m}_j, \pi_j), sig_s \rangle$

   Phase I: Message dissemination

10 Upon $\langle \textsc{send}, h', (\tilde{m}_i, \pi_i, i), sig_s \rangle$ arrival from $p_s$ do
11     if $\neg$isValid$\big(h', \{(\tilde{m}_i, \pi_i, i)\}, \{sig_s\}\big)$ then return        ▷ discard invalid messages
12     if $p_i$ already executed l. 15 or signed a msg from $p_s$ with hash $h'' \neq h'$ then return
13     store $\tilde{m}_i$ and $sig_s$ for $h'$
14     $sig_i \leftarrow \big(\mathsf{sign}_i(h'), i\big)$ ; store $sig_i$ for $h'$
15     broadcast $\langle \textsc{forward}, h', (\tilde{m}_i, \pi_i, i), \{sig_s, sig_i\} \rangle$

16 Upon $\langle \textsc{forward}, h', fragtuple_j, sigs_j = \{sig_s, sig_j\} \rangle$ arrival from $p_j$ do
17     if $\neg$isValid$\big(h', \{fragtuple_j\}, sigs_j\big)$ then return        ▷ discard invalid messages
18     if $p_i$ already signed a message from $p_s$ with hash $h'' \neq h'$ then return
19     store $sigs_j$ for $h'$
20     if $fragtuple_j \neq \bot$ then
21         $(\tilde{m}_j, \pi_j, j) \leftarrow fragtuple_j$ ; store $\tilde{m}_j$ for $h'$
22     if no forward message sent yet then
23         $sig_i \leftarrow \big(\mathsf{sign}_i(h'), i\big)$ ; store $sig_i$ for $h'$
24         broadcast $\langle \textsc{forward}, h', \bot, \{sig_s, sig_i\} \rangle$

   Phase II: Reaching Quorum and Termination

25 When $\left\{ \begin{array}{l} \exists h' : \big|\{\text{stored signatures for } h'\}\big| > \frac{n+t}{2} \wedge \big|\{\text{stored } \tilde{m}_j \text{ for } h'\}\big| \geq k \\ \wedge \text{ no message has been MBRB-delivered yet} \end{array} \right\}$ do
26     $m_i \leftarrow \mathsf{ECC}^{-1}(\tilde{m}_1, \ldots, \tilde{m}_n), \begin{cases} \text{where } \tilde{m}_j \text{ are taken from line 25;} \\ \text{when a fragment is missing use } \bot. \end{cases}$
27     $\big(h, (\tilde{m}'_j, \pi'_j, j)_j\big) \leftarrow$ computeFragMerkleTree($m_i$)
28     if $h' = h$ then
29         $sigs_h \leftarrow \{$all stored signatures for $h\}$
30         comm($v_1, \ldots, v_n$) where $v_j = \langle \textsc{bundle}, h, (\tilde{m}'_i, \pi'_i, i), (\tilde{m}'_j, \pi'_j, j), sigs_h \rangle$
31         MBRBdeliver($m_i$)

32 Upon $\langle \textsc{bundle}, h', (\tilde{m}'_j, \pi'_j, j), fragtuple'_i, sigs \rangle$ arrival from $p_j$ do
33     if $\neg$isValid$\big(h', \big\{(\tilde{m}'_j, \pi'_j, j), fragtuple'_i\big\}, sigs\big)$ then return   ▷ discard invalid msgs
34     if $|sigs| \leq \frac{n+t}{2}$ then return                          ▷ discard msgs with no quorum
35     store $(\tilde{m}'_j, \pi'_j, j)$ and $sigs$ for $h'$
36     if no bundle message has been sent yet $\wedge$ $fragtuple'_i \neq \bot$ then
37         $(\tilde{m}'_i, \pi'_i, i) \leftarrow fragtuple'_i$
38         store $(\tilde{m}'_i, \pi'_i, i)$ for $h'$
39         broadcast $\langle \textsc{bundle}, h', (\tilde{m}'_i, \pi'_i, i), \bot, sigs \rangle$
```

References

1 Timothé Albouy, Davide Frey, Ran Gelles, Carmit Hazay, Michel Raynal, Elad Michael Schiller, François Taïani, and Vassilis Zikas. Towards optimal communication Byzantine reliable broadcast under a message adversary. *CoRR*, abs/2312.16253, 2023. doi:10.48550/arXiv.2312.16253.

2 Timothé Albouy, Davide Frey, Michel Raynal, and François Taïani. Asynchronous Byzantine reliable broadcast with a message adversary. *Theor. Comput. Sci.*, 978:114110, 2023. doi:10.1016/J.TCS.2023.114110.

3 Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Balanced Byzantine reliable broadcast with near-optimal communication and improved computation. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, pages 399–417, 2022. doi:10.1145/3519270.3538475.

4 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

5 Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *CCS*, pages 2705–2721. ACM, 2021. doi:10.1145/3460120.3484808.

6 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *J. ACM*, 32(1):191–204, January 1985. doi:10.1145/2455.214112.

7 Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 218–238, New York, NY, 1990. Springer New York. doi:10.1007/0-387-34805-0_21.

8 Kartik Nayak, Ling Ren, Elaine Shi, Nitin H. Vaidya, and Zhuolun Xiang. Improved extension protocols for Byzantine broadcast and agreement. In *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179, pages 28:1–28:17, 2020. doi:10.4230/LIPIcs.DISC.2020.28.

9 Michel Raynal. Message adversaries. In *Encyclopedia of Algorithms*, pages 1272–1276. Springer, 2016. doi:10.1007/978-1-4939-2864-4_609.

10 Ron M. Roth. *Introduction to coding theory*. Cambridge University Press, 2006.

11 Nicola Santoro and Peter Widmayer. Time is not a healer. In *STACS 89*, pages 304–313, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. doi:10.1007/BFb0028994.

12 Nicola Santoro and Peter Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theor. Comput. Sci.*, 384(2-3):232–249, 2007. doi:10.1016/J.TCS.2007.04.036.

Brief Announcement: Solvability of Three-Process General Tasks

Hagit Attiya
Department of Computer Science, Technion, Haifa, Israel

Pierre Fraigniaud
IRIF – CNRS & Université Paris Cité, France

Ami Paz
LISN – CNRS & Université Paris-Saclay, France

Sergio Rajsbaum
Instituto de Matemáticas, UNAM, Mexico City, Mexico

Abstract

The topological view on distributed computing represents a task T as a relation Δ between the complex $\mathcal{I}$ of its inputs and the complex $\mathcal{O}$ of its outputs. A cornerstone result in the field is an elegant computability characterization of the solvability of colorless tasks in terms of $\mathcal{I}$, $\mathcal{O}$ and Δ. Essentially, *a colorless task is wait-free solvable if and only if there is a continuous map from the geometric realization of* $\mathcal{I}$ *to that of* $\mathcal{O}$ *that respects* Δ.

This paper makes headway towards providing an analogous characterization for general tasks, which are not necessarily colorless, by concentrating on the case of three-process inputless tasks. Our key contribution is identifying *local articulation points* as an obstacle for the solvability of general tasks, and defining a topological deformation on the output complex of a task T, which eliminates these points by splitting them, to obtain a new task T', with an adjusted relation Δ' between the input complex $\mathcal{I}$ and an output complex $\mathcal{O}'$ without articulation points. We obtain a new characterization of wait-free solvability of three-process general tasks: T *is wait-free solvable if and only if there is a continuous map from the geometric realization of* $\mathcal{I}$ *to that of* $\mathcal{O}'$ *that respects* Δ'.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Wait-free computing, lower bounds, topology

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.42

Related Version
Full Version: `https://hagit.net.technion.ac.il/files/2024/10/DISC2024BA-long.pdf`

Funding *Hagit Attiya*: partially supported by the Israel Science Foundation (22/1425).
Pierre Fraigniaud: partially supported by the ANR projects DUCAT (ANR-20-CE48-0006), FREDDA (ANR-17-CE40-0013), and QuData (ANR-18-CE47-0010).
Sergio Rajsbaum: Partially done while visiting IRIF-France and supported by the ANR project DUCAT (ANR-20-CE48-0006).

1 Introduction

For more than thirty years, distributed computing has been studied through the lens of topology, developing a deep understanding of the *solvability of tasks*. In this approach, a *simplex* represents a configuration as a set of vertexes, each representing the state of one process. In general, each vertex has an associated process *id*, sometimes referred to as its *color*. *Tasks* are triples $(\mathcal{I}, \mathcal{O}, \Delta)$, where $\mathcal{I}$ and $\mathcal{O}$ are *simplicial complexes* modelling the inputs and outputs of the task, and Δ is a relation specifying the possible valid outputs, $\Delta(\sigma)$, for each input simplex $\sigma \in \mathcal{I}$. For any initial configuration σ of $\mathcal{I}$, each process starts with an input vertex of σ colored by its ID, and must decide on an output vertex with its color, such that the vertices decided by the processes form a simplex τ of $\Delta(\sigma)$.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 42; pp. 42:1–42:7
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

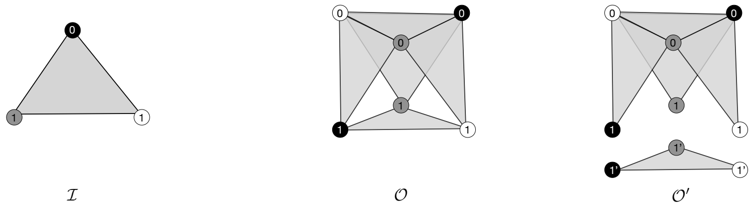


Figure 1 *The majority consensus task: input complex (left), output complex (center) and output complex after splitting (right).*

A major contribution of the topological approach is a set of novel impossibility results and algorithms for specific tasks, through fundamental characterizations of solvable tasks in various distributed computing models. Beyond telling us what is solvable and what is not, characterization results tell us *what makes tasks unsolvable*, indicating the obstructions to solvability, and sometimes pointing how these obstructions can be avoided.

The focus of this work is on *wait-free* protocols for solving a task in a read/write shared memory system. A cornerstone result for this model is the *asynchronous computability theorem* (ACT) of Herlihy and Shavit [18] (see also [14, Theorem 11.2.1]). The remarkable insight of this theorem is that a protocol solving a task in this model corresponds to a color-preserving simplicial mapping from a *chromatic subdivision* of the input complex into allowable outputs in the output complex.

As stated, however, the ACT does not provide us with a direct connection to standard topology notions relating the topology of the input and the output complexes. Such a characterization is so far known only for *colorless* tasks [7, 15]. A colorless task is defined only in terms of input and output values, regardless of the number of processes involved, and regardless of which process has a particular input or output value; accordingly, $\mathcal{I}$ and $\mathcal{O}$ consist of sets of values, without process IDs. Well-known examples are the consensus task [10] and its generalization to set consensus [8]. Colorless tasks are simpler to analyze and in particular, they have an elegant computability characterization in terms of their input and output complexes [14, 17]: *a colorless task is wait-free solvable if and only if there is a continuous map from* $|\mathcal{I}|$ *to* $|\mathcal{O}|$ *respecting* Δ. Recall that $|K|$ denotes the geometric realization of a simplicial complex.

A similar characterization for *general* tasks, which are not necessarily colorless, has eluded researchers. General tasks that cannot be stated as colorless tasks are sometimes called *chromatic*. Several such tasks have been studied, notably renaming [3]. A simple example is the *majority consensus* task. In this weaker form of binary consensus, it is allowed to decide different values (when processes do not all start with the same input), but only if more processes decide 0 than 1. Figure 1 illustrates this task for a single three-process input configuration: two processes start with 1 and the other with 0. Tasks whose input complex contains a single facet are called *inputless* in the literature. Impossibility results are usually achieved using inputless versions of tasks, which is also the case in our examples, and we concentrate on such tasks in this work. In our figures, processes are identified by black, grey and white colors, and their respective inputs are inside the vertices, and with the analogous convention for output values, the output complex, where the respective output values are inside vertices.

Researchers have tried to characterize the solvability of general tasks in terms of continuous maps, analogous to the colorless characterization. The value of such a characterization would not only be due to its direct nature from the input complex to the output complex, but also due to the direct connection to topology: continuous maps between spaces.

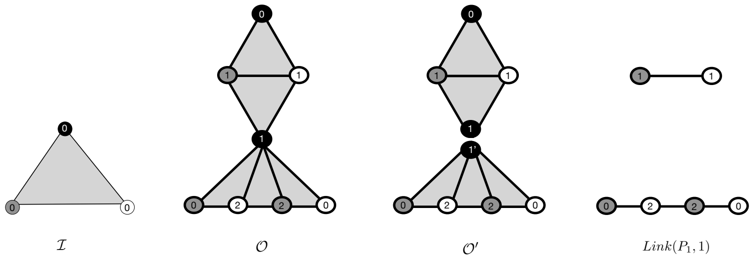


Figure 2 *The hourglass task: input complex (left), output complex (center left), output complex after splitting (center right) and the link of the local articulation point (right).*

The quest for such a characterization has failed, because *it is not true* that a chromatic task is solvable if and only if there is a continuous map from the input to the output complex. This has been demonstrated with the *hourglass* task (Figure 2) in [14, Section 11.1]. The hourglass task is an inputless task, where each process running solo decides on 0. Process P_0 (black) running concurrently with either P_1 or P_2 can also (in addition to their solo values) decide on their respective vertexes, with output 1. While P_1 and P_2 running concurrently can additionally decide their respective vertices with value 2. When all three run concurrently, any triangle is a valid output simplex. Despite there being a continuous map from the input to the output complex respecting the input/output relation for the hourglass task, the task is unsolvable. The same holds for the majority consensus task, as proved in our paper.

2 Our Results

This paper proves a *necessary and sufficient* condition for the solvability of colored *three-process* inputless tasks, $(\mathcal{I}, \mathcal{O}, \Delta)$, directly from the input complex, not to the output complex, $\mathcal{O}$, since this is impossible, but to an output complex $\mathcal{O}'$, easily derived from $\mathcal{O}$.

Our study goes through identifying a notion of a *local articulation point* (LAP) in the output complex: a vertex whose neighborhood, its *link* in the topological parlance, is disconnected. Figure 2(right) depicts the link of a vertex in the hourglass task, which is a graph consisting of two connected components: the output edges compatible with the vertex of P_1 deciding 1.

We show a novel method for dealing with each local articulation point by *splitting* the output complex around it, eventually creating a *link-connected* output complex $\mathcal{O}'$. See the right-hand side of Figure 1, for splitting the majority consensus task, and Figure 2, for splitting the hourglass task. This yields a task T', with the same input complex $\mathcal{I}$ as the original task and an adjusted relation Δ' between $\mathcal{I}$ and $\mathcal{O}'$. We show that the solvability of the original general task is equivalent to the solvability of the task derived from T by eliminating all the LAPs.

▶ **Theorem 1** (informal). *T is solvable if and only if T' is solvable.*

When T is inputless, this implies that it is wait-free solvable if and only if there is a continuous map from the geometric realization of $\mathcal{I}$ to that of $\mathcal{O}'$, which respects Δ'.

▶ **Theorem 2** (informal). *A general inputless task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is solvable if and only if there is a continuous map from $|\mathcal{I}|$ to $|\mathcal{O}'|$ carried by Δ'.*

The following necessary condition is useful when considering specific tasks:

▶ **Corollary 3.** *A general task $(\mathcal{I}, \mathcal{O}, \Delta)$ is not solvable if there is an edge $\{x, x'\} \in \mathcal{I}$ such that for every $y \in \Delta(x)$ and $y' \in \Delta(x')$, any path from y to y' in $\Delta(\{x, x'\})$ goes through a LAP.*

This corollary can be used to prove the impossibility of solving the majority consensus task, the hourglass task, and of the *pinwheel* task, which we define in the full paper.

Three-Process Tasks as a Stepping-Stone for Future Investigation. The restriction to three processes could be seen as a limitation of our results, and indeed it is a stepping-stone for future investigation. However, there are several reasons for concentrating on this case.

First, in this case there is no need to use algebraic topology, making the paper accessible to a wider audience.

Second, the case of three processes has played an important role in past research, because it is the smallest dimension where topological properties beyond graph connectivity appear. In this case, two types of obstructions – local articulation points and contractibility – are neatly identifiable. In smaller dimensions, e.g., for two-process tasks, if the output complex has a local articulation point then it is also not connected (in the graph-theoretic sense), and hence, it is not solvable. Indeed, for two-process general tasks, there is a characterization based on continuous maps (a consequence of [14, Theorem 2.5.2], analogous to the seminal one for message passing models [5]). On the other hand, in dimensions higher than 2, i.e., with four or more processes, a disconnected link may be connected (in the graph-theoretic sense).

Finally, undecidability results such as [11, 16] are proved using colorless *loop agreement* tasks [15, 16, 21], defined using an output complex $\mathcal{O}$, and a *loop* in it. Roughly, each process starts on one of *three* distinguished vertexes of the loop; if they start on the same vertex, they decide on this vertex; if they start on two distinct distinguished vertexes, then they decide vertices belonging to the same edge along the path linking their starting vertexes; finally, if they start on all three distinguished vertices, then they can decide vertexes belonging to any simplex of $\mathcal{O}$. Like all colorless tasks, loop agreement is defined independently of the number of processes. Notice however that it is defined on *two-dimensional* input and output complexes, and hence, all the arguments are essentially in the *three-process* case. Moreover, some results [15, Theorem 5.4] do not generalize to more than three processes. Note that our approach is different from the one used in [11], where the undecidability of three-process tasks was proved by a reduction from the *contractibility problem* to the task-solvability problem. The contractibility problem asks whether a given loop of a simplicial complex can be continuously transformed into a point, a problem which is known to be undecidable even for 2-dimensional simplicial complexes (see, e.g., [23]). Gafni and Koutsoupias prove the reduction by showing that contractibility is undecidable for the special case of chromatic complexes and loops of length 3. To do so, they first show that the contractibility problem is undecidable for link-connected two-dimensional complexes. We instead transform the output complex to be link-connected, and can then argue directly about colorless tasks.

3 Discussion and Related Work

This paper provides a new characterization for wait-free solvability of general three-process tasks, which is based on splitting *local articulation points*. We prove that the task T is wait-free solvable if and only if there is a continuous map from the geometric realization of the input complex of T to the geometric realization of the deformed output complex, which respects the deformed task mapping.

Our characterization exposes two types of obstructions to solvability: The first, which exists only in chromatic tasks, are local articulation points; these obstructions can be effectively detected (and removed). The second is identical to the one that exists for colorless tasks, namely, the existence of a continuous map from the geometric realization of $\mathcal{I}$ to that of $\mathcal{O}'$ carried by Δ'. The locality of the former obstruction makes it an ideal target for extension-based impossibility proofs [2,4]. The latter obstruction is known to be undecidable [11,15], as it is closely related to the topological notion of loop contractibility.

Link connectivity has showed up in previous papers about chromatic tasks, starting with the work of Herlihy and Shavit [18]. Nevertheless, our paper is the first to identify the precise role of link connectivity, by concentrating on the special case of three process.

There are two previous approximations to continuous characterization. First, when assuming link connectivity, there are sufficiency results for general tasks ([22] and [14, Section 11.5]), without a matching necessary condition. Another related notion are *continuous tasks* [12], which have an input/output specification that is a continuous function between the geometric realizations of the input and output complex. The characterization is that a task is solvable if and only if there exists a *chromatic function* (a notion introduced in this paper) from the input complex $\mathcal{I}$ to the output complex $\mathcal{O}$ respecting the task specification. Our characterization is in contrast more explicit about the obstructions (since it exposes the role of local articulation points), and establishes a direct connection with colorless solvability (after removing articulation points, colored and colorless solvability are the same).

Havlicek [13] have studied the goal of identifying *computable obstructions* to wait-free solvability, taking into account that any such mechanism must be necessarily incomplete, due to the known undecidability of this problem [11, 15]. The mechanism presented by Havlicek can nevertheless find obstructions for consensus and other tasks, but only those related to homology groups. It would be interesting to extend it to deal with the type of link connectivity obstructions studied here.

The full paper presents new chromatic versions of consensus, *majority consensus*, approximate agreement, *majority approximate agreement*, and set agreement, *pinwheel task*, that may be of interest on their own. They are obtained by removing some *output* triangles. Notice that removing output simplexes is the opposite of what is done in the condition-based approach [20], where instead *input* simplexes are removed to obtain an easier task, instead of a harder one.

Removing output simplexes to obtain our new tasks perturbs the symmetry of the original tasks, so that the chromatic version can no longer be defined without referring to ids. It would be interesting to derive a systematic way of transforming any colorless task in this way. It would be also interesting to consider the same idea but for any number of processes, n, and investigate the simplexes than need to be removed to obtain a decidable obstruction. That is, removing some output simplexes of $k \leq n$ processes, while otherwise leaving the task unchanged. For what value of k the obstruction becomes decidable?

Pseudospheres [1] are a succinct mathematical notation that was used in the topological approach of distributed computing to state that any process can take any value. Our examples leave intact all the pseudospheres of dimension 1, while destroying those of dimension 2. It would be nice to generalize the examples to higher dimensions (and more than three processes).

Our splitting deformation draws upon work on modelling of real-world objects, used in computer-aided design (CAD) [19]. There is a long line of papers studying splitting deformations mostly of two and three-dimensional simplicial complexes, because these are the dimensions of most graphics applications (and for technical reasons, as discussed in,

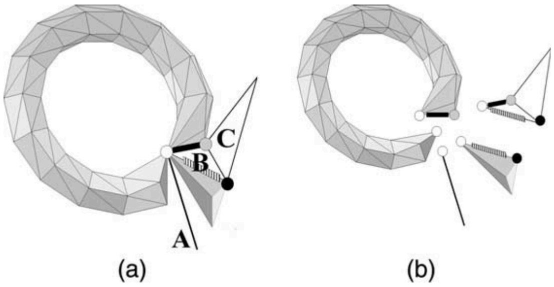


Figure 3 *[9, Figure 1] An example (a) of a non-manifold object (described by a 3D simplicial complex made of tetrahedra, triangles, and edges) with a dangling edge (A) and a dangling surface formed by two triangles (B) and (C) and its decomposition (b) into "simple" components.*

e.g. [6]), although there is also work on higher dimensional complexes. The same splitting we do has been used (e.g. [9, Fig.1], replicated in Figure 3), but not to fix a disconnected link; instead, the interest has been in doing additional splittings, even of edges, because the goal in this research line is to decompose a non-manifold complex into an assembly of manifolds, or at least into components that belong to simpler, well-understood class of complexes where efficient data structures are known.

References

1 Luis Alberto. Pseudospheres: combinatorics, topology and distributed systems. *Journal of Applied and Computational Topology*, 2024. doi:10.1007/s41468-023-00162-5.

2 Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili, and Leqi Zhu. Why extension-based proofs fail. *SIAM Journal on Computing*, 52(4):913–944, 2023. doi:10.1137/20M1375851.

3 Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990. doi:10.1145/79147.79158.

4 Hagit Attiya, Armando Castañeda, and Sergio Rajsbaum. Locally solvable tasks and the limitations of valency arguments. In *24th International Conference on Principles of Distributed Systems (OPODIS)*, volume 184, pages 18:1–18:16, 2020. doi:10.4230/LIPIcs.OPODIS.2020.18.

5 Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed 1-solvable tasks. *Journal of algorithms*, 11(3):420–440, 1990. doi:10.1016/0196-6774(90)90020-F.

6 Dobrina Boltcheva, David Canino, Sara Merino Aceituno, Jean-Claude Léon, Leila De Floriani, and Franck Hétroy. An iterative algorithm for homology computation on simplicial shapes. *Computer-Aided Design*, 43(11):1457–1467, 2011. Solid and Physical Modeling 2011. doi:10.1016/j.cad.2011.08.015.

7 Elizabeth Borowsky, Eli Gafni, Nancy A. Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Comput.*, 14(3):127–146, 2001. doi:10.1007/PL00008933.

8 Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, 1993. doi:10.1006/INCO.1993.1043.

9 Leila De Floriani, Mostefa M. Mesmoudi, Franco Morando, and Enrico Puppo. Decomposing non-manifold objects in arbitrary dimensions. *Graphical Models*, 65(1):2–22, 2003. doi:10.1016/S1524-0703(03)00006-7.

10 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

11 Eli Gafni and Elias Koutsoupias. Three-processor tasks are undecidable. *SIAM Journal on Computing*, 28(3):970–983, 1998. doi:10.1137/S0097539796305766.

12 Hugo Rincon Galeana, Sergio Rajsbaum, and Ulrich Schmid. Continuous tasks and the asynchronous computability theorem. In *13th Innovations in Theoretical Computer Science Conference, ITCS*, pages 73:1–73:27, 2022. doi:10.4230/LIPICS.ITCS.2022.73.

13 John Havlicek. Computable obstructions to wait-free computability. *Distributed Computing*, 13(2):59–83, 2000. doi:10.1007/s004460050068.

14 Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology.* Morgan Kaufmann, 2013.

15 Maurice Herlihy and Sergio Rajsbaum. The decidability of distributed decision tasks. In *Proceedings of the 29th annual ACM symposium on Theory of computing*, pages 589–598, 1997.

16 Maurice Herlihy and Sergio Rajsbaum. A classification of wait-free loop agreement tasks. *Theoretical Computer Science*, 291(1):55–77, 2003. doi:10.1016/S0304-3975(01)00396-6.

17 Maurice Herlihy, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. From wait-free to arbitrary concurrent solo executions in colorless distributed computing. *Theor. Comput. Sci.*, 683:1–21, 2017. doi:10.1016/J.TCS.2017.04.007.

18 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999. doi:10.1145/331524.331529.

19 Annie Hui and Leila De Floriani. A two-level topological decomposition for non-manifold simplicial shapes. In *Proceedings of the 2007 ACM symposium on Solid and Physical Modeling*, pages 355–360, 2007. doi:10.1145/1236246.1236297.

20 Achour Mostéfaoui, Sergio Rajsbaum, and Michel Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *J. ACM*, 50(6):922–954, 2003. doi:10.1145/950620.950624.

21 Vikram Saraph and Maurice Herlihy. The relative power of composite loop agreement tasks. In *19th International Conference on Principles of Distributed Systems, OPODIS*, pages 13:1–13:16, 2015. doi:10.4230/LIPICS.OPODIS.2015.13.

22 Vikram Saraph, Maurice Herlihy, and Eli Gafni. An algorithmic approach to the asynchronous computability theorem. *J. Appl. Comput. Topol.*, 1(3-4):451–474, 2018. doi:10.1007/S41468-018-0014-4.

23 John C. Stillwell. *Classical topology and combinatorial group theory*, volume 72 of *Graduate Texts in Mathematics.* Springer-Verlag, New York, 1980.

Brief Announcement: Unifying Partial Synchrony

Andrei Constantinescu
ETH Zürich, Switzerland

Diana Ghinea
ETH Zürich, Switzerland

Jakub Sliwinski
ETH Zürich, Switzerland

Roger Wattenhofer
ETH Zürich, Switzerland

Abstract

The distributed computing literature considers multiple options for modeling communication. Most simply, communication is categorized as either synchronous or asynchronous. Synchronous communication assumes that messages get delivered within a publicly known timeframe and that parties' clocks are synchronized. Asynchronous communication, on the other hand, only assumes that messages get delivered eventually. A more nuanced approach, or a middle ground between the two extremes, is given by the *partially synchronous model*, which is arguably the most realistic option. This model comes in two commonly considered flavors:

(i) The Global Stabilization Time (GST) model: after an (unknown) amount of time, the network becomes synchronous. This captures scenarios where network issues are transient.

(ii) The Unknown Latency (UL) model: the network is, in fact, synchronous, but the message delay bound is unknown.

This work formally establishes that any time-agnostic property that can be achieved by a protocol in the UL model can also be achieved by a (possibly different) protocol in the GST model. By time-agnostic, we mean properties that can depend on the order in which events happen but not on time as measured by the parties. Most properties considered in distributed computing are time-agnostic. The converse was already known, even without the time-agnostic requirement, so our result shows that the two network conditions are, under one sensible assumption, equally demanding.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases partial synchrony, unknown latency, global stabilization time

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.43

1 Introduction

Distributed computing systems underpin a vast array of contemporary technological advancements, ranging from cloud computing platforms to blockchain networks. These systems rely on protocols to ensure consistency and reliability even when faced with challenges such as message delays and node failures. A cornerstone of designing robust protocols lies in understanding the communication model assumed by the distributed system. Within the distributed computing literature, the synchronous and asynchronous communication models remain the two best-established paradigms. The synchronous model assumes a publicly known upper bound Δ on message delays and that parties hold synchronized clocks. This idealized setting facilitates the design of elegant round-based protocols that often achieve very high resilience thresholds. However, the synchronous model exhibits a fundamental limitation: any deviation from the assumed message delay bound Δ can render synchronous protocols entirely ineffective, potentially leading to complete breakdowns of the protocols.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 43; pp. 43:1–43:7
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The asynchronous model, on the other hand, only assumes that messages get delivered eventually. This inherent flexibility empowers the asynchronous model to support protocols that can gracefully adapt to any network conditions. However, asynchronous protocols typically exhibit lower resilience thresholds compared to their synchronous counterparts, and even achieving agreement when parties might crash is impossible without randomization [3]. Hence, neither of these two extremes perfectly captures real-world systems: the synchronous model's assumptions are too strong, while the asynchronous model is too pessimistic. In this work, we are concerned with a middle ground between the two: the *partially synchronous* model, a nuanced paradigm that bridges the gap between the two, introduced by Dwork, Lynch, and Stockmeyer [2]. The work of [2] proposes two definitions for the partially synchronous model, described below (see the next section for the formal definitions).

The Global Stabilization Time (GST) model. This variant acknowledges that there might be periods of unpredictable delays due to network congestion or outages but also assumes that these disruptions eventually resolve and the system stabilizes. [2] explains how this intuition can be faithfully captured with a simple model, known as the Global Stabilization Time (GST) model: there is an unknown "Global Stabilization Time" T after which the system behaves synchronously for a publicly-known message delivery bound Δ. In particular, there is a publicly known amount of time Δ such that every message sent at time t is delivered by time $\max(\mathsf{t}, \mathsf{T}) + \Delta$.

The Unknown Latency (UL) model. In this variant, the system is, in fact, always synchronous: there is a value Δ such that every message sent by time t is delivered by time $\mathsf{t} + \Delta$. However, as opposed to the synchronous model, the value of Δ is unknown to the protocol.

The relationship between the two. The two models are conjectured to be equivalent, in the sense that any property that can be achieved by a protocol in one can also be achieved in the other. In fact, there is an elegant folklore reduction from the UL model to the GST model, presented in [1], which we explain next. Consider a protocol Π achieving some property X in the UL model. Let us run Π in the GST model, where a value of Δ is provided and guaranteed to hold eventually, but Π ignores it. Consider any execution ε of Π in this setting: the model ensures that there is a time T such that all messages in ε sent at time t get delivered by time $\max(\mathsf{t}, \mathsf{T}) + \Delta$. Hence, in ε all messages get delivered within time $\mathsf{T} + \Delta$, so from the parties' perspective, they might just as well be running in UL with the unknown bound on message delay being $\mathsf{T} + \Delta$. Hence, ε is also a legal execution of Π in UL. As a result, the set of executions of Π in GST is a subset of its set of executions in UL. Since Π satisfies X in UL, it also satisfies X in GST.

The same blog post [1] also explains the reverse direction: a protocol designed for the GST model can be transformed into an equivalent protocol for the UL model, but only if it satisfies a certain property, namely, that the protocol's guarantees are still maintained if the assumed value of Δ changes dynamically. This way, one may increment the assumed Δ whenever a timeout of the protocol expires, and eventually, the assumed Δ exceeds the real one. However, as [1] points out, assuming this property is with loss of generality. Whether the converse holds is still an open question.[1]

[1] The incorrect proof also implicitly assumes the time-agnostic property which we introduce in the next paragraph.

Our contribution. In this work, we answer this question in the affirmative under the relatively minor technical assumption of only considering "time-agnostic" properties. A protocol property is *time-agnostic* if whether it holds for a given execution of a protocol can only depend on the relative order in which events happened, but not on time as measured by the parties in the execution. We note that most properties considered in distributed computing are indeed time-agnostic; e.g., whether some consensus protocol satisfies a given agreement conditions. Bounds on message complexity can also be accommodated, but the same is not true about running time. Additionally, we will only show our result assuming that the environment provides a global perfect clock to the parties, that is their only way of telling time. On a similar note, we consider randomized protocols, but do not consider probabilistic properties, such as "with probability at least 0.5 all parties terminate". We leave a formal argument considering imperfect clocks and probabilistic properties for future work. On the other hand, our proof works in adversarial settings, i.e., crashes or byzantine behavior. The key idea in our proof is that, instead of estimating the actual value of Δ in the UL model, like in the argument of [1], we continuously slow down the parties' clocks. This is achieved by the parties applying a wrapper function on top of the time measurements returned by the system clock. This way, the parties simulate running the GST protocol with a continuously increasing value of Δ, which eventually exceeds the actual unknown Δ of the UL model, hence allowing the guarantees of the GST protocol that we are running to apply.

2 Preliminaries

We consider a fixed set of n parties in a network, where links model communication channels. The parties are running a (possibly randomized) protocol over the network. For each party, the protocol is specified by a state transition diagram, where a party's state is defined by its local variables. The initial state of a party is then defined by any initial inputs and random coins. The transitions are deterministic (but may depend on the party's random coins). Without loss of generality, a party's transition to another state is triggered by the receipt of a message, or specific changes in time (e.g., waiting a predefined amount of time). State changes are instantaneous and include all required local computations and the corresponding sending of messages (i.e., these instructions are executed atomically). The receipt of messages, on the other hand, will be controlled by the *message system*, which we discuss below.

Messages. Messages are held in a global *message system*: this maintains a set containing tuples (P_s, P_r, m, c), where P_s is the sender of the message, P_r is a receiver of the message, m is the content of the message, and c is a unique identifier assigned by the message system. The message system is controlled by the adversary and may decide when to deliver these messages (subject to the constraints of the communication model). For simplicity, we assume that the message system keeps delivering messages even after the receivers have terminated (if the protocol allows it) or crashed. Otherwise, claims of the form "eventually all messages get delivered within Δ time units" would not be meaningful for terminated receivers.

Global clock. We assume that parties have access to a common *global clock* denoted by $\textsc{Clock}$, which is their only source of time. Abstractly, $\textsc{Clock}$ is represented by an increasing and continuous function $\textsc{Clock} : \mathbb{R}_{\geqslant 0} \to \mathbb{R}_{\geqslant 0}$ that maps real time to *system time*. In particular, at real time t, the parties can atomically query the global clock to read off a "system time" of $\textsc{Clock}(t)$. Neither the parties nor the adversary have access to the actual definition of the function $\textsc{Clock}$. Instead, they can only use the global clock as an oracle to receive the current *system time*. Depending on the environment, the system time may coincide with real time, in which case $\textsc{Clock}$ is $\textsc{RealClock}(t) = t$, but this is not necessarily the case.

Protocol execution models. A protocol execution model M captures all requirements and guarantees of the environment under which a protocol runs. For our scope, communication in M always happens through message passing as already described. Moreover, M specifies a global clock function $\textsc{Clock}$ that the parties use to tell time. Other aspects of the execution environment can appear as part of the guarantees of M, such as message delay bounds or other timing constraints. Two examples of such models M are $\mathrm{GST}(\Delta, \textsc{Clock})$ and $\mathrm{UL}(\Delta, \textsc{Clock})$, formally introduced below. Note that the guarantees of a fixed model M are concrete: e.g., messages are delivered within Δ time units for a fixed Δ; in contrast, often in the literature, models usually refer to families of models (in this particular case, parameterized by Δ). Last but not least, M specifies the power of the adversary. On top of controlling the scheduler within the model's timing constraints, the adversary might, for instance, make parties crash, fail to send certain messages or deviate from the protocol arbitrarily (i.e., byzantine behavior). Model M should specify precisely which faults are possible and under what circumstances (e.g., if the adversary is adaptive, computationally bounded, and how many parties it can corrupt). The parties are not aware of the clock function used: this is supplied by the environment as an oracle, with no access to its implementation. More abstractly, a model M specifies for each protocol Π its set of legal executions ε, defined next.

Executions. Consider a protocol Π running in a model M where parties measure time using function $\textsc{Clock}$. An execution of Π is defined by the parties' initial states and a (possibly infinite) collection of events, denoted by $\textsc{events}(\varepsilon)$. Each event in $\textsc{events}(\varepsilon)$ is a tuple (t, $\textsc{ReceivedMsgs}$, P, q, $\textsc{SentMsgs}$) signifying that, at system time t (i.e., as observed by the parties using function $\textsc{Clock}$), party P received the (possibly empty) multiset of messages $\textsc{ReceivedMsgs}$ from the message system, P's state became q (possibly the same state it was already in), and P sent the (possibly empty) multiset of messages $\textsc{SentMsgs}$ to the message system. We say that a message $\textsc{msg} = (P_s, P_r, m, c)$ was sent at system time t in execution ε if $\textsc{events}(\varepsilon)$ contains some event $(t, \textsc{ReceivedMsgs}, P_s, q, \textsc{SentMsgs})$ with $\textsc{msg} \in \textsc{SentMsgs}$. Similarly, we say that a message $\textsc{msg} = (P_s, P_r, m, c)$ was received at system time t in execution ε if $\textsc{events}(\varepsilon)$ contains some event $(t, \textsc{ReceivedMsgs}, P_r, q, \textsc{SentMsgs})$ with $\textsc{msg} \in \textsc{ReceivedMsgs}$. Note that a message sent/received at system time t is sent/received at real time $\textsc{Clock}^{-1}(t)$. We have made the deliberate choice to timestamp executions in system time as this is the perspective that parties perceive them from. This will allow us to map between executions with different clock functions in our main result.

The GST model. The GST model has as parameters a clock function $\textsc{Clock}$ that the environment provides to the parties to tell the time when running a protocol, and Δ, to be supplied to protocols designed for the model when instantiated for a specific Δ. We write $\mathrm{GST}(\Delta, \textsc{Clock})$ for the model instantiated with specific parameters Δ and $\textsc{Clock}$. The model guarantees that, for every protocol Π and every execution ε of Π in the model, there exists a time T measured in real time such that every message in ε sent at real time t is received by real time $\max(t, T) + \Delta$. The model can be altered to give the adversary more power than controlling the scheduler; e.g., to corrupt parties.

The UL model. The UL model has as parameters a clock function $\textsc{Clock}$ that the environment similarly provides to the parties, and Δ, not to be supplied to protocols designed for the model. We write $\mathrm{UL}(\Delta, \textsc{Clock})$ for the model instantiated with specific parameters Δ and $\textsc{Clock}$. The model guarantees that, for every protocol Π and every execution ε of Π in the model, any message in ε sent at real time t is received by real time $t + \Delta$. This model can also be altered to give more power to the adversary.

Protocol properties. We define a protocol property as a set of allowed executions; e.g., the property that all parties eventually terminate, or that they produce some outputs. A protocol achieves a property in a model M if all its legal executions in that model satisfy the property, i.e., are in the set of executions allowed by the property. Note that modeling certain properties this way is non-trivial, as executions alone do not contain, e.g., who are the byzantine parties and when they were corrupted. However, even such properties can be modeled: executions may contain changes of states that do not follow from the protocol's state transition to model parties misbehaving, or one can modify executions to include corruption events to make the process more transparent. In this paper, we are concerned with time-agnostic properties, defined next. We call two executions $\varepsilon, \varepsilon'$ *equivalent* if they differ only in the timestamps of the events and agree on the relative order of the events. A property X is *time-agnostic* if for any two equivalent executions $\varepsilon, \varepsilon'$ it holds that $\varepsilon \in X \iff \varepsilon' \in X$.

Augmented models. Our result will be very general: we will consider an arbitrary protocol Π designed for the GST model, instantiated with a publicly-known eventual message delay bound of 1, that satisfies a given time-agnostic property in $\mathrm{GST}(1, \textsc{RealClock})$. We will show how Π can be transformed into a protocol Π' that only depends on Π that satisfies the same property in $\mathrm{UL}(\Delta, \textsc{RealClock})$, irrespective of the value of Δ. Moreover, if we augment both the GST model and the UL model with the same kind of additional power for the adversary, the same statement holds, with the same proof. For simplicity, in the following, we assume the basic models, but we note that we also get the result for a plethora of more interesting fault settings, e.g., byzantine faults and crashes.

3 Our Reduction

This section presents the proof of our main result, stated below.

▶ **Theorem 1.** *Any time-agnostic property that can be achieved by a protocol in the GST model can also be achieved by a protocol in the UL model.*

As previously mentioned, the key idea behind our reduction will be slowing down time. Given a protocol Π achieving a time-agnostic property X in $\mathrm{GST}(1, \textsc{RealClock})$, we construct a protocol Π' such that any execution ε' of Π' in $\mathrm{UL}(\Delta, \textsc{RealClock})$ for some Δ unbeknownst to the protocol is equivalent to a legal execution ε of Π in $\mathrm{GST}(1, \textsc{RealClock})$, hence also achieving property X. Protocol Π' will simulate running Π with a modified system clock that continuously slows down, so that equal intervals of time measured in the simulated system will represent longer and longer spans of real time. Moreover, we need that the modified system clock eventually gets arbitrarily slow. This way, since Π is designed to have property X if, once sufficient time passes, every message gets delivered within 1 unit of system time, this will eventually be the case: the clock gets slow enough for 1 unit of system time to correspond to a span of real time exceeding the unknown message delay bound.

More specifically, Π' will simulate Π running with system clock $\textsc{SlowClock} : \mathbb{R}_{\geqslant 0} \to \mathbb{R}_{\geqslant 0}$ given by $\textsc{SlowClock}(t) = \sqrt{t}$ (any increasing function whose derivative tends to 0 as $t \to \infty$ will suffice). Whenever a party in the simulated Π queries the global clock and the answer would have normally been (real time) t, Π' replaces the answer with $\textsc{SlowClock}(t)$: from the perspective of the simulated Π, the system clock is $\textsc{SlowClock}$.

The first lemma below shows the required result assuming that Π is running standalone but with system clock $\textsc{SlowClock}$. The second lemma lifts it to the protocol Π' that runs with system clock $\textsc{RealClock}$, but simulates Π running with system clock $\textsc{SlowClock}$. A short discussion of why this implies Theorem 1 follows.

▶ **Lemma 2.** *Consider a protocol Π and a legal execution ε of Π in UL(Δ, $\textsc{SlowClock}$). Then, ε is a legal execution of Π in GST$(1, \textsc{RealClock})$.*

Proof. Consider an execution ε of Π in $\mathrm{UL}(\Delta, \textsc{SlowClock})$, which guarantees that any message sent at real time t is delivered by real time $t+\Delta$. From the perspective of the parties, however, time is measured using $\textsc{SlowClock}$, so in ε, the parties observe that any message sent at system time $\textsc{SlowClock}(t)$ is delivered by system time $\textsc{SlowClock}(t+\Delta)$. Let us consider $\textsc{SlowClock}(t+\Delta) - \textsc{SlowClock}(t) = \sqrt{t+\Delta} - \sqrt{t}$ to understand how the message delay observed by the parties evolves with t. Taking the derivative, the function is strictly decreasing with t, so the observed network delay gets smaller and smaller as time passes. Subsequently, let us find a bound t_0 on t such that starting at real time t_0, the observed network delay is bounded by 1; i.e., let us solve $\sqrt{t+\Delta} - \sqrt{t} \leqslant 1$. If $\Delta < 1$, this happens for $t \geqslant 0$. Otherwise, $\Delta \geqslant 1$, and this happens for $t \geqslant \frac{1}{4}(\Delta - 1)^2$. Hence, starting at real time $\frac{1}{4}(\max\{1, \Delta\} - 1)^2$, the observed (system) network delay is bounded by 1. Writing the same in terms of system time, starting at system time $T := \sqrt{\frac{1}{4}(\max\{1,\Delta\} - 1)^2} = \frac{1}{2}(\max\{1, \Delta\} - 1)$, the system network delay is bounded by 1. In particular, this means that a message sent at system time t in ε is delivered by system time $\max\{t, T\} + 1$. Hence, ε could just as well be an execution of Π in $\mathrm{GST}(1, \textsc{RealClock})$ with global stabilization time T because the parties and the adversary are unaware of the clock function used. ◀

▶ **Lemma 3.** *If protocol Π achieves a time-agnostic property X in* GST$(1$, $\textsc{RealClock})$, *there is a protocol Π' depending only on Π that achieves X in UL(Δ, $\textsc{RealClock}$) $\forall \Delta \geqslant 0$.*

Proof. In protocol Π' parties run protocol Π but apply the function $\textsc{SlowClock} : \mathbb{R}_{\geqslant 0} \to \mathbb{R}_{\geqslant 0}$ as a wrapper over the global clock's responses to the queries. In particular, whenever a party queries the global clock in Π and the time returned is t, the party evaluates $\textsc{SlowClock}(t)$ and takes this as the answer instead. Every execution ε' of Π' in some model M corresponds to an equivalent execution ε of Π in M where the clock function provided by the environment is composed with $\textsc{SlowClock}$. Namely, every event e present in ε and ε' is timestamped t in ε and $\textsc{SlowClock}(t)$ in ε'.

Hence, for any $\Delta \geqslant 0$, any legal execution ε' of Π' in $\mathrm{UL}(\Delta, \textsc{RealClock})$ corresponds to a legal execution ε of Π in $\mathrm{UL}(\Delta, \textsc{SlowClock})$ that is equivalent to ε'. By Lemma 2, ε is also a legal execution of Π in $\mathrm{GST}(1, \textsc{RealClock})$. Since Π achieves property X in $\mathrm{GST}(1, \textsc{RealClock})$, it follows that $\varepsilon \in X$, and hence, since X is time-agnostic, $\varepsilon' \in X$. Since ε' was an arbitrary execution of Π' in $\mathrm{UL}(\Delta, \textsc{RealClock})$ and $\Delta \geqslant 0$ was arbitrary, it follows that Π' satisfies property X in $\mathrm{UL}(\Delta, \textsc{RealClock})$ for all $\Delta \geqslant 0$. ◀

Proof of Theorem 1. If $\boldsymbol{\Pi}$ denotes the set of all protocols, a protocol designed for the GST model is, in fact, a protocol family $\Pi : \mathbb{R}_{\geqslant 0} \to \boldsymbol{\Pi}$, one for each potential value of the publicly-known eventual message delay bound. Π achieves a property in $\mathrm{GST}(\Delta, \textsc{RealClock})$ for some $\Delta \geqslant 0$ iff all executions of $\Pi(\Delta)$ achieve this property in $\mathrm{GST}(\Delta, \textsc{RealClock})$.

Let Π be a protocol achieving a time-agnostic property X in the GST model. We only need that $\Pi(1)$ satisfies X in $\mathrm{GST}(1, \textsc{RealClock})$: applying Lemma 3 to $\Pi(1)$, we get that Π' satisfies X in $\mathrm{UL}(\Delta, \textsc{RealClock})$ for all $\Delta \geqslant 0$, implying the conclusion. ◀

References

1 Ittai Abraham. Flavours of partial synchrony, 2019. URL: https://decentralizedthoughts.github.io/2019-09-13-flavours-of-partial-synchrony/.

2 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988. doi:10.1145/42282.42283.

3 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.

Brief Announcement: The Expressive Power of Uniform Population Protocols with Logarithmic Space

Philipp Czerner
Technical University of Munich, Germany

Vincent Fischer
Technical University of Munich, Germany

Roland Guttenberg
Technical University of Munich, Germany

Abstract

Population protocols are a model of computation in which indistinguishable mobile agents interact in pairs to decide a property of their initial configuration. Originally introduced by Angluin et. al. in 2004 with a constant number of states, research nowadays focuses on protocols where the space usage depends on the number of agents. The expressive power of population protocols has so far however only been determined for protocols using $o(\log n)$ states, which compute only semilinear predicates, and for $\Omega(n)$ states. This leaves a significant gap, particularly concerning protocols with $\Theta(\log n)$ or $\Theta(\text{polylog}\, n)$ states, which are the most common constructions in the literature. In this paper we close the gap and prove that for any $\varepsilon > 0$ and $f \in \Omega(\log n) \cap \mathcal{O}(n^{1-\varepsilon})$, both uniform and non-uniform population protocols with $\Theta(f(n))$ states can decide exactly $\mathsf{NSPACE}(f(n) \log n)$.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Population Protocols, Uniform, Expressive Power

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.44

Related Version *Full Version*: https://arxiv.org/abs/2408.10027 [15]

1 Introduction

Population protocols are a model of computation in which indistinguishable mobile agents randomly interact in pairs to decide whether their initial configuration satisfies a given property. The decision is taken by *stable consensus*; eventually all agents agree on whether the property holds or not, and never change their mind again. While originally introduced to model sensor networks [4], population protocols are also very close to chemical reaction networks [23], a model in which agents are molecules and interactions are chemical reactions.

Originally agents were assumed to have a finite number of states [4, 5, 6], however many predicates then provably require at least $\Omega(n)$ time to decide [21, 7, 1], as opposed to recent breakthroughs of $\mathcal{O}(\log n)$ time using $\mathcal{O}(\log n)$ number of states (in some cases even $\mathcal{O}(\log \log n)$ states) for important tasks like leader election [9] and majority [19]. Limiting the number of states to logarithmic is important in most applications, especially the chemical reaction setting, since a linear in n number of states would imply the unrealistic number of approximately 10^{23} different chemical species. Therefore most recent literature focusses on the polylogarithmic time and space setting, and determines time-space tradeoffs for various important tasks like majority [3, 1, 2, 22, 8, 19], leader election [1, 22, 9] or estimating/counting the population size [20, 16, 10, 17, 18].

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 44; pp. 44:1–44:7
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This leads to the interesting open problem of characterizing the class of predicates which can be computed in polylogarithmic time using a logarithmic or polylogarithmic number of states. There is however a fundamental problem with working on this question: Despite the focus on $\mathcal{O}(\log n)$ number of states in recent times, the expressive power for this number of states (regardless of time) is still unknown.

More precisely, there is a gap in the existing literature: protocols with $f(n) \in \Omega(n)$ states are known to have expressive power $\mathsf{SNSPACE}(n \log f(n))$ [14], i.e. symmetric predicates in $\mathsf{NSPACE}(n \log f(n))$, while a subclass of protocols with $o(\log n)$ states can only compute semilinear predicates [6, 14]. The latter result applies only to *uniform* population protocols, i.e. protocols where the transitions are independent of the size of the population.

However, many constructions in the literature have e.g. $\Theta(\log n)$ or $\Theta(\operatorname{polylog} n)$ states. This important case is not covered by the existing results. To the best of our knowledge, the only research in this direction is [12], where the expressive power is characterised for $\operatorname{polylog}(n)$ number of states for a similar model – not population protocols themselves. Most importantly, their results do not lead to a complete characterization for $\Theta(\log n)$ states since they lose some log factors in their characterization of $\operatorname{polylog}(n)$.

In this paper, we fill the gap by proving that for functions $f(n) \in \Omega(\log n) \cap \mathcal{O}(n^{1-\varepsilon})$, where $\varepsilon > 0$, population protocol with $f(n)$ states compute exactly $\mathsf{SNSPACE}(f(n) \cdot \log n)$, i.e. the symmetric predicates computable by a non-deterministic Turing machine using $\mathcal{O}(f(n) \cdot \log n)$ space. This result applies to both uniform and non-uniform protocols. (The function f needs to fulfil some technical conditions.)

With this result, the expressive power of uniform population protocols is characterised in all cases, and for non-uniform protocols it is characterised in the case of $\Omega(\log n)$ states. (A slight gap between $\mathcal{O}(n^{1-\varepsilon})$ and $\Omega(n)$ remains.)

2 Preliminaries

▶ **Definition 1.** *A* protocol scheme $\mathcal{P}$ *is a 5-tuple* $(Q, \Sigma, \delta, I, O)$ *of*

- *a (not necessarily finite) set of states* Q*,*
- *a finite input alphabet* Σ*,*
- *a (partial) transition function* $\delta : Q \times Q \to Q \times Q$*,*
- *an injective input mapping* $I \colon \Sigma \to Q$*,*
- *an output mapping* $O \colon Q \to \{0, 1\}$*.*

A *configuration* of $\mathcal{P}$ is a finite multiset $C \in \mathbb{N}^Q$, which represents a collection of agents with states in Q. A step $C \to C'$ in $\mathcal{P}$ occurs by choosing two agents from C and letting them interact via δ, i.e. if their states are p, q in C, then their new states in C' will be $\delta(p, q)$.

We write $\to^*$ for the reflexive and transitive closure of $\to$, and say that a configuration C' is *reachable* from C if $C \to^* C'$. The input to $\mathcal{P}$ consists of a multiset $w \in \mathbb{N}^\Sigma$. Every input w can be mapped to its corresponding *initial* configuration by applying I to every letter in w

A configuration C is a *b-consensus* for $b \in \{0, 1\}$ if $O(q) = b$ for all q such that $C(q) \neq 0$, i.e. if every state which occurs in the configuration has output b. A configuration C is *stable with output* b if every configuration C' reachable from C is a b-consensus.

A *run* ρ is an infinite sequence of configurations $\rho = (C_0, C_1, \dots)$ such that $C_i \to C_{i+1}$ for all $i \in \mathbb{N}$. A run is *fair* if for all configurations C which occur infinitely often in ρ, i.e. such that there are infinitely many i with $C_i = C$, every configuration C' reachable from C occurs infinitely often in ρ. A run has *output* b if some configuration C_i along the run is stable with output b (and hence all C_j for $j \geq i$ are also stable with output b).

An input $w \in \mathbb{N}^\Sigma$ has *output* b if every fair run starting at its corresponding initial configuration $\hat{I}(w)$ has output b. The protocol scheme $\mathcal{P}$ *computes* a predicate $\varphi\colon \mathbb{N}^\Sigma \to \{0,1\}$ if every input w has output $\varphi(w)$.

Let us provide an example which also shows how to treat infinite sets Q.

▶ **Example 2.** Consider $Q := \{0\} \cup \{2^i \mid i \in \mathbb{N}\}$, and define $\delta(2^i, 2^i) = (2^{i+1}, 0)$. Let $\Sigma = \{x\}$, and let $x \mapsto 2^0$ be the input mapping. Then a configuration is initial if every agent is in state 2^0. Intuitively this protocol will eventually end up with the binary representation of the number of agents. Namely each transition preserves the total sum of all agents' values, and every transition increases the number of agents in 0, so this protocol in fact always reaches a terminal configuration.

Regarding the infinite state space, intuitively the protocol uses $\lfloor \log n \rfloor + 2$ states, namely $\lfloor \log n \rfloor + 1$ powers of two and 0. The other states cannot be reached with n agents.

Accordingly we now define the state complexity of a protocol scheme. A state $q \in Q$ is *coverable* from some initial configuration C_0 if there exists a configuration C reachable from C_0 which fulfils $C(q) > 0$. The *state complexity* $S(n)$ of $\mathcal{P}$ for n agents is the number of states $q \in Q$ which are coverable from some initial configuration with n agents.

▶ **Example 3.** In the scheme of Example 2, let C_n be the unique initial configuration with n agents, i.e. $C_n(2^0) = n$ and $C_n(q) = 0$ otherwise. For $n \geq 2$, the states coverable from C_n are exactly $\{0\} \cup \{2^i \mid i \leq \log n\}$. Hence the state complexity is $S(n) = \lfloor \log n \rfloor + 2$.

As defined so far, protocol schemes are not necessarily computable. Hence actual population protocols require some uniformity condition.

▶ **Definition 4.** *A* uniform population protocol $\mathcal{P} = (Q, \Sigma, \delta, I, O)$ *is a protocol scheme together with a bijection* $Q \to \{0,1\}^*$ *to represent* Q *via binary strings, such that the functions* δ, I, O *are computable by linear space Turing-machines (TMs).*

We remark that "linear space" then in terms of our n, the number of agents, is $\mathcal{O}(\log S(n))$ space (since the input of the machine is a representation of a state).

In the literature on uniform population protocols, e.g. [13, 14, 20, 16], often agents are defined as TMs and states hence automatically assumed to be represented as binary strings. We avoid talking about the exact implementation of a protocol via TMs because it introduces an additional logarithm in the number of states and potentially confuses the reader, while most examples are clearly computable.

▶ **Example 5.** In the protocol scheme of Example 2 we represent states by the binary representation of the exponent. Clearly incrementing natural numbers or setting the number to a fixed value are possible by a linear space TM, hence this is a uniform population protocol.

Next we define a more general class of population protocols, which we call weakly uniform. This class includes all known population protocols, and our results also hold for this class, which shows that having a different protocol for every n does not strengthen the model.

▶ **Definition 6.** *A* finite population protocol *is a protocol scheme with a finite set* Q.

A population protocol $\mathcal{P}$ *is an infinite family* $(\mathcal{P}_n)_{n\in\mathbb{N}} = (Q_n, \Sigma, \delta_n, I_n, O_n)_n$ *of finite population protocols. The state complexity for inputs of size* n *is* $S(n) := |Q_n|$.

$\mathcal{P}$ *is* weakly uniform *if there exist TMs* M_δ, M_I, M_O *using* $\mathcal{O}(S(n))$ *space which compute* δ_n, I_n *and* O_n, *respectively, taking* n *as additional input.*

The configurations of $\mathcal{P}$ with n agents are exactly the configurations of $\mathcal{P}_n$ with n agents, and accordingly the semantics of steps, runs and acceptance are inherited from $\mathcal{P}_n$.

The protocol for a given population size n is allowed to differ completely from the protocol for $n-1$ agents, as long as TMs are still able to evaluate transitions, input and output. Usually this is not fully utilised, with the most common case of a non-uniform protocol being that $\log n$ is encoded into the transition function [19].

Clearly uniform population protocols are weakly uniform. Namely let $\mathcal{P} = (Q, \Sigma, \delta, I, O)$ be a protocol scheme. Then for every $n \in \mathbb{N}$ we let Q_n be the set of states coverable by some initial configuration with n agents, similar to the definition of state complexity, and define $\mathcal{P}_n := (Q_n, \Sigma, \delta_n|_{Q_n^2}, I, O|_{Q_n})$, where $f|_A$ is the restriction of f to inputs in A. This protocol family computes the same predicate, and is weakly-uniform with the same state complexity.

Next we define the complexity classes for our main result. Let $f\colon \mathbb{N} \to \mathbb{N}$ be a function. f is space-constructible if there exists a TM M which computes f using $\mathcal{O}(f(n))$ space. Given a space-constructible function $f\colon \mathbb{N} \to \mathbb{N}$, we denote by $\mathsf{NSPACE}(f(n))$ the class of predicates computable by a non-deterministic Turing-machine in $\mathcal{O}(f(n))$ space, and by $\mathsf{SNSPACE}(f(n))$ the class of symmetric (i.e. only depending on the count of letters) predicates in $\mathsf{NSPACE}(f(n))$. Similarly, let $\mathsf{UPP}(f(n))$ be the class of predicates computable by uniform population protocols with $\mathcal{O}(f(n))$ space, and $\mathsf{WUPP}(f(n))$ be the class of predicates computable by weakly-uniform population protocols with $\mathcal{O}(f(n))$ space.

3 Main Result

We give a characterisation for the expressive power of both uniform and weakly uniform population protocols with $f(n)$ states, where $f \in \Omega(\log n) \cap \mathcal{O}(n^{1-\varepsilon})$, for some $\varepsilon > 0$. For technical reasons, we must place a few limitations on $f(n)$ (see the full paper [15] for details). We will refer to a function f fulfilling these requirements as *reasonable*.

Our bound applies to uniform and weakly uniform protocols. As mentioned in the previous section, the latter includes, to the best of our knowledge, all non-uniform constructions from the literature.

▶ **Theorem 7.** *Let $\varepsilon > 0$ and let $f \in \Omega(\log n) \cap \mathcal{O}(n^{1-\varepsilon})$ be reasonable. Then*

$$\mathsf{UPP}(f(n)) = \mathsf{WUPP}(f(n)) = \mathsf{SNSPACE}(f(n) \cdot \log n)$$

Proof. This will follow from Proposition 8 and Theorem 9. ◀

In particular, we have $\mathsf{UPP}(\log n) = \mathsf{WUPP}(\log n) = \mathsf{SNSPACE}(\log^2 n)$.

▶ **Proposition 8** (Upper Bound). *Let $\varepsilon > 0$ and let $f \in \Omega(\log n) \cap \mathcal{O}(n^{1-\varepsilon})$ be space-constructible. Then*

$$\mathsf{UPP}(f(n)) \subseteq \mathsf{WUPP}(f(n)) \subseteq \mathsf{SNSPACE}(f(n) \log n)$$

Proof (sketch). $\mathsf{UPP}(f(n)) \subseteq \mathsf{WUPP}(f(n))$ is trivial/was explained in Section 2. $\mathsf{WUPP}(f(n)) \subseteq \mathsf{SNSPACE}(f(n) \log n)$ can be shown using a reduction to a reachability problem in the configuration graph as in [11]. ◀

Our main contribution is the proof of the lower bound:

▶ **Theorem 9** (Lower Bound). *Let $\varepsilon > 0$ and let $f \in \Omega(\log n) \cap \mathcal{O}(n^{1-\varepsilon})$ be reasonable. Then*

$$\mathsf{SNSPACE}(f(n) \log n) \subseteq \mathsf{UPP}(f(n))$$

Proof (sketch). We construct a uniform population protocol $\mathcal{P} = (Q, \Sigma, \delta, I, O)$, simulating a counter machine with $|\Sigma|$ input counters using space $\mathcal{O}(2^{f(n)\log n})$, which is equivalent to a $\mathcal{O}(f(n)\log n)$ space-bounded Turing machine.

Initialisation. Our first goal is to reach a configuration where the number of agents n is known. By this we mean, that we want to have $\lfloor \log n \rfloor + 1$ uniquely indentifiable "counter agents", each of which stores one bit of the binary representation of n. We use a similar approach as in Example 2 to achieve this:

$$\begin{aligned}
&(\mathsf{Ctr}, i, 1), (\mathsf{Ctr}, i, 1) \mapsto (\mathsf{Ctr}, i+1, 1), (\mathsf{Ctr}, i, 0) && \text{for } i \in \mathbb{N} \\
&(\mathsf{Ctr}, i, 0), (\mathsf{Ctr}, i, b) \mapsto (\mathsf{Ctr}, i, b), (\mathsf{Ldr}, i+1) && \text{for } i \in \mathbb{N}
\end{aligned} \qquad \langle \mathsf{counter} \rangle$$

Here $(\mathsf{Ctr}, i, 1)$ encodes that the i-th bit in the binary representation of n is set, while $(\mathsf{Ctr}, i, 0)$ represents an unset bit. The first transition is analogous to Example 2, but instead of simply sending the second agent to state 0, it also remembers which bit it represents. The second transition gets rid of additional agents storing the same bit.

Among the remaining agents we now want to elect one leader, who knows how many bits n has. We will refer to all agents which are neither counters nor a leader as *free*:

$$\begin{aligned}
&(\mathsf{Ldr}, i), (\mathsf{Ldr}, j) \mapsto (\mathsf{Ldr}, j), \mathsf{Free} && \text{for } i, j \in \mathbb{N}, i \leq j \\
&(\mathsf{Ldr}, i), (\mathsf{Ctr}, j, b) \mapsto (\mathsf{Ldr}, j), (\mathsf{Ctr}, j, b) && \text{for } i, j \in \mathbb{N}, i \leq j
\end{aligned} \qquad \langle \mathsf{leader} \rangle$$

The first transition here is a standard leader election, the second informs the leader of the number of bits required to store n.

At some point a configuration will be reached, where the binary counting and leader elections have been completed. Note that there is no way of telling for certain when this is the case. For now, we will assume that we have reached such a configuration and describe how to solve this problem later on.

Once n is known, we gain the ability to loop over all agents: each of the counter agents stores an additional, initially unset, bit. Every agent stores a marker flag. The Leader can then apply operations to all agents by sequentially interacting with them and setting the marker flag. Each time the leader interacts with an agent which has the marker flag unset, it increments the second value stored in the counter agents. If at some point both values match, then, as the first value is n, all of the agents must have been marked. In particular this allows the leader to check if an agent with a certain state does not exist. Normally this is quite difficult for population protocols, as agents in the queried state might not take part in any interactions for an arbitrarily long time.

Simulating Counter Machines. Often, when population protocols need to simulate some type of counter, either a unary [5], or binary encoding [12], is used. Neither approach works for us, as we need to be able to count up to $2^{f(n)\log n}$, but a unary encoding with n agents is bounded by n, and a binary encoding with $f(n)$ distinguishable digits is bounded by $2^{f(n)}$. Instead we use a mixed-radix positional encoding with the base $b_i \in \Omega(\frac{n}{f(n)})$ for every digit i. To achieve this, the leader evenly divides the remaining free agents into $\Omega(f(n))$ groups, each encoding one digit. Recall that the leader can detect when no free agents remain, so it will know when this process is finished. Within each digit unary counting is used, that is, each agent in that digit stores one counter bit and the overall value of the digit is the sum of all counter bits. The commands of the counter machine involve manipulating these digits, by either incrementing, or decrementing the encoded values, as well as checking whether they are zero. For the latter, the leader again uses the ability to detect whether a state is present in the population.

Resets. The counter machine simulation described in the previous section relies on the looping and absence checks enabled by the data structures set up during initialisation. However, there is no way of being certain that the initialisation has finished. We solve this by raising a dirty flag each time a transition from the initialisation phase occurs. When seen by the leader, this will trigger a reset, where the leader will move all agents back to state Free, once again relying on being able to count the number of agents. When the last reset occurs, the counter agents must encode the correct value of n, and the leader is thus able to iterate over all agents. Care must be taken s.t. other agents do not interact with agents in Free while the reset is ongoing, e.g. when only half of the agents have moved to Free, and the others are still some intermediate states. ◀

References

1 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L. Rivest. Time-space trade-offs in population protocols. In *SODA 2017*, pages 2560–2579. SIAM, 2017. doi:10.1137/1.9781611974782.169.

2 Dan Alistarh and Rati Gelashvili. Recent algorithmic advances in population protocols. *SIGACT News*, 49(3):63–73, 2018. doi:10.1145/3289137.3289150.

3 Dan Alistarh, Rati Gelashvili, and Milan Vojnovic. Fast and exact majority in population protocols. In *PODC*, pages 47–56. ACM, 2015. doi:10.1145/2767386.2767429.

4 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *PODC 2004*, pages 290–299. ACM, 2004. doi:10.1145/1011767.1011810.

5 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. In *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2006. doi:10.1007/11864219_5.

6 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Comput.*, 20(4):279–304, 2007. doi:10.1007/S00446-007-0040-2.

7 Amanda Belleville, David Doty, and David Soloveichik. Hardness of computing and approximating predicates and functions with leaderless population protocols. In *ICALP*, volume 80 of *LIPIcs*, pages 141:1–141:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.ICALP.2017.141.

8 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. Time-space trade-offs in population protocols for the majority problem. *Distributed Comput.*, 34(2):91–111, 2021. doi:10.1007/S00446-020-00385-0.

9 Petra Berenbrink, George Giakkoupis, and Peter Kling. Optimal time and space leader election in population protocols. In *STOC*, pages 119–129. ACM, 2020. doi:10.1145/3357713.3384312.

10 Petra Berenbrink, Dominik Kaaser, and Tomasz Radzik. On counting the population size. In *PODC*, pages 43–52. ACM, 2019. doi:10.1145/3293611.3331631.

11 Michael Blondin, Javier Esparza, and Stefan Jaax. Expressive power of broadcast consensus protocols. In *CONCUR*, volume 140 of *LIPIcs*, pages 31:1–31:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.CONCUR.2019.31.

12 Olivier Bournez, Johanne Cohen, and Mikaël Rabie. Homonym population protocols. *Theory Comput. Syst.*, 62(5):1318–1346, 2018. doi:10.1007/S00224-017-9833-2.

13 Ioannis Chatzigiannakis, Othon Michail, Stavros Nikolaou, Andreas Pavlogiannis, and Paul G. Spirakis. Passively mobile communicating logarithmic space machines. *CoRR*, abs/1004.3395, 2010. arXiv:1004.3395.

14 Ioannis Chatzigiannakis, Othon Michail, Stavros Nikolaou, Andreas Pavlogiannis, and Paul G. Spirakis. Passively mobile communicating machines that use restricted space. *Theor. Comput. Sci.*, 412(46):6469–6483, 2011. doi:10.1016/J.TCS.2011.07.001.

15 Philipp Czerner, Vincent Fischer, and Roland Guttenberg. The expressive power of uniform population protocols with logarithmic space, 2024. arXiv:2408.10027.

16 David Doty and Mahsa Eftekhari. Efficient size estimation and impossibility of termination in uniform dense population protocols. In *PODC*, pages 34–42. ACM, 2019. doi:10.1145/3293611.3331627.

17 David Doty and Mahsa Eftekhari. A survey of size counting in population protocols. *Theor. Comput. Sci.*, 894:91–102, 2021. doi:10.1016/J.TCS.2021.08.038.

18 David Doty and Mahsa Eftekhari. Dynamic size counting in population protocols. In *SAND*, volume 221 of *LIPIcs*, pages 13:1–13:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.SAND.2022.13.

19 David Doty, Mahsa Eftekhari, Leszek Gasieniec, Eric E. Severson, Przemyslaw Uznanski, and Grzegorz Stachowiak. A time and space optimal stable population protocol solving exact majority. In *FOCS*, pages 1044–1055. IEEE, 2021. doi:10.1109/FOCS52979.2021.00104.

20 David Doty, Mahsa Eftekhari, Othon Michail, Paul G. Spirakis, and Michail Theofilatos. Brief announcement: Exact size counting in uniform population protocols in nearly logarithmic time. In *DISC*, volume 121 of *LIPIcs*, pages 46:1–46:3. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.DISC.2018.46.

21 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. In *DISC*, volume 9363 of *Lecture Notes in Computer Science*, pages 602–616. Springer, 2015. doi:10.1007/978-3-662-48653-5_40.

22 Robert Elsässer and Tomasz Radzik. Recent results in population protocols for exact majority and leader election. *Bull. EATCS*, 126, 2018. URL: http://bulletin.eatcs.org/index.php/beatcs/article/view/549/546.

23 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Nat. Comput.*, 7(4):615–633, 2008. doi:10.1007/S11047-008-9067-Y.

Brief Announcement: Best-Possible Unpredictable Proof-Of-Stake

Lei Fan ✉
Shanghai Jiao Tong University, China

Jonathan Katz ✉
Google, Washington DC, USA
University of Maryland, College Park, MD, USA

Zhenghao Lu ✉
Shanghai Jiao Tong University, China

Phuc Thai ✉
Sky Mavis, Ho Chi Minh City, Vietnam

Hong-Sheng Zhou ✉
Virginia Commonwealth University, Richmond, VA, USA

Abstract

The proof-of-stake (PoS) protocols aim to reduce the unnecessary computing power waste seen in Bitcoin. Various practical and provably secure designs have been proposed, like Ouroboros Praos (Eurocrypt 2018) and Snow White (FC 2019). However, the essential security property of unpredictability in these protocols remains insufficiently explored. This paper delves into this property in the cryptographic setting to achieve the "best possible" unpredictability for PoS.

We first present an impossibility result for *all* PoS protocols under the *single-extension* design framework, where each honest player extends one chain per round. The state-of-the-art permissionless PoS protocols (e.g., Praos, Snow White, and more), are all under this single-extension framework. Our impossibility result states that, if a single-extension PoS protocol achieves the best possible unpredictability, then this protocol cannot be proven secure unless more than 73% of stake is honest. To overcome this impossibility, we introduce a new design framework called *multi-extension* PoS, allowing each honest player to extend *multiple* chains using a greedy strategy in a round. This strategy allows us to construct a class of PoS protocols that achieve the best possible unpredictability. It is noteworthy that these protocols can be proven secure, assuming a much smaller fraction (e.g., 57%) of stake to be honest.

2012 ACM Subject Classification Computing methodologies → Distributed computing methodologies

Keywords and phrases blockchain, consensus, proof-of-stake, unpredictability

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.45

Related Version *Full Version*: `https://eprint.iacr.org/2021/660`

Funding Phuc Thai and Hong-Sheng Zhou were supported in part by NSF grant CNS-1801470.

Acknowledgements This project was conducted during Phuc Thai's time as a PhD student at Virginia Commonwealth University.

1 Introduction

Cryptocurrencies like Bitcoin [13] have proven to be a phenomenal success. These protocols are executed by a **large-size** peer-to-peer network of nodes using the proof-of-work (PoW) mechanism [9, 2]. They provide a trustworthy, append-only, and always-available public ledger, facilitating the implementation of a global payment system (e.g., Bitcoin) or a global computer (e.g., Ethereum). However, the PoW-based consensus requires substantial

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 45; pp. 45:1–45:7
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

computing power. Utilizing alternative resources like *coins (also known as stake)* to secure a blockchain is desirable. If successful, the new system would be environmentally friendly, as it would not rely on extensive computing power for security. Several attempts have been made, with PoS mechanisms widely discussed in the cryptocurrency community (e.g., [1, 12, 15, 4]). In a PoS-based blockchain protocol, players must prove ownership of a specified number of stakes; only those who can provide such proofs are permitted to participate in maintaining the blockchain. Compared with PoW mechanisms, the computational cost of finding solutions in PoS mechanisms is very "cheap."

Early PoS designs (e.g., [1, 12, 15, 4]) and PoW-based designs, such as the original Bitcoin, were initially crafted in an *ad hoc* style. However, the contemporary trend leans towards a more rigorous approach where security concerns are precisely defined, and the designed protocols undergo mathematical analysis. Notable contributions include the work by Garay et al. [10] and Pass et al. [14], analyzing the PoW-based blockchain in Bitcoin within the *cryptographic setting*. The analysis demonstrated that the Bitcoin blockchain can achieve crucial security properties, such as common prefix, chain quality, and chain growth. Indeed, research efforts have also been devoted to PoS-based and Bitcoin-like consensus, as seen in [8, 7, 3]. Nevertheless, these protocols are vulnerable to attacks due to predictability.

Intuitively, predictability in a protocol implies that certain players are aware they will be selected to generate blockchain blocks before actually doing so. Brown-Cohen et al. [5] explored the predictability of PoS in incentive-driven scenarios, where players may deviate from the protocol for higher profits. The power of predictability can be exploited by attackers to reduce the difficulty or cost of incentive-driven attacks like selfish-mining [5] or bribery [3]. Therefore, it is crucial for a PoS protocol to minimize predictability and mitigate the risks of these attacks. Ideally, a PoS protocol should aim for the *best possible* unpredictability, enabling effective counteraction of predictability-based attacks. Achieving this goal ensures the maintenance of blockchain fairness and incentivizes honest players to participate in the protocol.

Our first result is that we formally define (the best possible) unpredictability in the cryptographic setting. We assert that a protocol achieves the best possible unpredictability if it only allows players to predict whether they can generate the next block, and nothing more. Based on the definition of the best possible unpredictability, we identify an interesting impossibility for a class of PoS protocols following a *single-extension* design framework. Existing provably secure Bitcoin-like PoS protocols (e.g., [8, 7, 3]) are all within the single-extension framework. Finally, to overcome the impossibility, we develop a novel D-distance-greedy strategy in the multi-extension framework, which allows us to design a provably secure Bitcoin-like PoS protocol.

2 Security Model

The security of Bitcoin-like PoW-based protocols has been rigorously investigated by Garay et al. [10] and then by Pass et al. [14] in the cryptographic setting.

The execution of a PoS blockchain protocol. Following Canetti's formulation[6], we present an abstract model for a PoS blockchain protocol Π in the hybrid world of the semi-synchronous network communication functionality, the random oracles, and certain initialization functionality, similarly drawn from [14].

We consider the execution of blockchain protocol Π that is directed by an environment $\mathcal{Z}(1^\kappa)$, where κ is a security parameter. A necessary condition in all common blockchain systems is that all players agree on the first, i.e., the *genesis block*, which consists of the identities (e.g., public keys) and the stake distribution of the players. The environment $\mathcal{Z}$ can "manage" players through an adversary $\mathcal{A}$ that can dynamically corrupt honest players. In any round r, each PoS-player $P \in \mathcal{P}$, with a local state st, receives a message from $\mathcal{Z}$, and potentially receives messages from other players. Then, it executes the protocol, broadcasts a message to other players, and updates its local state. Note that the network is under the control of $\mathcal{A}$, meaning that $\mathcal{A}$ is responsible for delivering all messages sent by players. Let $\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}}$ be a random variable denoting the joint $\textsc{view}$ of all players in the above protocol execution; note that this joint view fully determines the execution. More details of the formulation can be found in the full version of our paper.

Block and blockchain basics. A *blockchain* $\mathcal{C}$ consists of a sequence of ℓ concatenated blocks $B_0 \| B_1 \| B_2 \| \cdots \| B_\ell$, where $\ell \geq 0$ and B_0 is the genesis block. We use $\mathsf{len}(\mathcal{C})$ to denote *blockchain length*, i.e., the number of blocks in blockchain $\mathcal{C}$; and here $\mathsf{len}(\mathcal{C}) = \ell$. (Note that since all chains must consist of the genesis block, we do not count it as part of the chain's length.) We use sub blockchain (or subchain) for referring to a segment of a chain; here for example, $\mathcal{C}[j, m]$, with $j \geq 0$ and $m \leq \ell$ would refer to a sub blockchain $B_j \| \cdots \| B_m$. We use $\mathcal{C}[i]$ to denote the i-th block, B_i in blockchain $\mathcal{C}$; here i denotes the *block height* of B_i in chain $\mathcal{C}$. If blockchain $\mathcal{C}$ is a prefix of another blockchain $\mathcal{C}_1$, we write $\mathcal{C} \preceq \mathcal{C}_1$.

Chain growth, common prefix, and chain quality. Previously, several fundamental security properties for Bitcoin-like PoW-based blockchain protocols have been defined: *chain growth* [11], *common prefix* [10, 14], and *chain quality* [10]. Intuitively, the chain growth property states that the chains of honest players should grow linearly to the number of rounds. The common prefix property indicates the consistency of any two honest chains except the last κ blocks. The chain quality property, characterized by the parameter $\mu \in (0, 1)$, aims to indicate the ratio of contributions from honest players that are contained in a sufficiently long and continuous part of an honest chain, is at least μ.

Unpredictability. At a high level, predictability means that (certain) protocol players are aware that they will be selected to generate blocks of the blockchain, *before* they actually generate the blocks. We investigate the unpredictability in the cryptographic setting.

Consider a malicious player $P \in \mathcal{P}$ at round r. Let $\textsc{view}^r$ be the view of all players at round r, and $\mathcal{C}^r$ be the best (valid) chain of all players in $\textsc{view}^r$. At round r, the adversary $\mathcal{A}$ attempts to predict if the (malicious) player P can extend the best chain at a future round r', where $r' > r$. Let $z_P^{r'} \in \{0, 1\}$ be a prediction: $z_P^{r'} = 1$ means that $\mathcal{A}$ predicts that player P can extend the best chain at round r'. Now we introduce another random variable $\bar{z}_P^{r'}$ to indicate if P indeed can extend the best chain at round r' (as the adversary predicated at an early-round r) or not. Let $\textsc{view}^{r'}$ be the view of all players at round r', and $\mathcal{C}^{r'}$ be the best valid chain of all players in $\textsc{view}^{r'}$. We set $\bar{z}_P^{r'} = 1$ if there exists a chain $\mathcal{C} = \mathcal{C}^{r'} \| B$ in $\textsc{view}$ with a block B generated by player P at round r', otherwise we set $\bar{z}_P^{r'} = 0$.

Consider a view $\textsc{view}$, protocol round r, and a malicious player P. For a prediction $z_P^{r'}$ where $r' > r$, we define the predicate $\mathsf{predictable}$ to be true if the prediction $z_P^{r'}$ accurately predicts whether or not player P can generate a new chain at round r' that is 2 blocks longer than the longest chain at round r. (In any PoS protocol, all players can *always* predict whether or not they can generate the next block, so we consider 2 blocks.) More concretely, we define $\mathsf{predictable}(\textsc{view}, P, r, r', z_P^{r'}) = 1$ if and only if the following three conditions hold: *(i)* $r' > r$; *(ii)* $\mathsf{len}(\mathcal{C}^{r'}) + 1 - \mathsf{len}(\mathcal{C}^r) = 2$; and *(iii)* $z_P^{r'} = \bar{z}_P^{r'}$.

▶ **Definition 1** (The best possible unpredictability)**.** *Consider a blockchain protocol* Π*. We say protocol* Π *achieves the best possible unpredictability if for all* PPT $\mathcal{Z}, \mathcal{A}$*, for any malicious player* P *at any round* r*, we have,*

$$\Pr\left[\ \textsc{view} \leftarrow \mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}}; (r', z_P^{r'}) \leftarrow \mathcal{A}(P, r, \textsc{view}^r)\ \middle|\ \mathsf{predictable}(\textsc{view}, P, r, r', z_P^{r'}) = 0\ \right] > 1 - \mathsf{neg}(\kappa),$$

where $\mathsf{neg}(\cdot)$ *is a negligible function.*

3 An Impossibility Result

In this section, we present an impossibility result for a class of PoS protocols in the *single-extension* PoS framework. Intuitively, for Bitcoin-like PoS protocol in the *single-extension* framework, in each round, each honest player identifies only a single "best chain," and then extends this chain. The formal definition of this framework is presented in the full version of our paper. We remark that the state-of-the-art PoS protocols (e.g., [7, 8, 3]) can be categorized as single-extension PoS protocols.

Then we present an impossibility result for single-extension PoS protocols. Concretely, consider a PoS protocol in the single-extension framework, we can show that, if the PoS protocol achieves the best possible unpredictability, then the protocol cannot maintain security properties, such as the common prefix, when honest players control less than 73% of the stake. Let N be the number of players and ρ be the fraction of malicious players in the protocol execution. Let p be the probability that a player can extend a chain in a round. The probability that honest players extend a chain in a round is $\alpha = 1 - (1 - p)^{N\cdot(1-\rho)}$. Similarly, the probability that the adversary extends a given chain is $\beta = 1 - (1 - p)^{N\cdot\rho} \approx \frac{\rho}{1-\rho} \cdot \alpha$, if p is sufficiently small. The impossibility theorem is stated as follows, and the proof can be found in the full version.

▶ **Theorem 2.** *Consider a single-extension PoS protocol* Π *achieves the best possible unpredictability. If* $\alpha < e \cdot \beta$*, where* $e = 2.72$*, then* Π *cannot achieve common prefix property.*

4 Greedy Strategies: How to overcome the impossibility

In the previous section, we have obtained the impossibility of single-extension PoS protocols. In this section, we will introduce greedy strategies that follow a *multi-extension* framework. In these strategies, honest players are allowed to extend multiple chains that are "close" to each other. Our protocol can achieve the best possible unpredictability while requiring a much smaller fraction (e.g., 57%) of honest stake to achieve security properties.

Specifically, we allow the players to take a *greedy* strategy to extend the chains in a protocol execution: instead of extending a single best chain (i.e., the longest chain), the players are allowed to extend *a set of best chains*, expecting to extend the best chain faster. This is possible because extending chains in a PoS protocol is "very cheap." We remark that the set of best chains should be carefully chosen; otherwise, the protocol may not be secure. In our greedy strategy, the honest player extends the set of chains that share the same common prefix after removing the last few blocks. With this strategy, the security of the protocol is guaranteed.

Distance-greedy strategies. Consider a protocol execution. In each player's local view, there are multiple chains, which can be viewed as a tree. Concretely, the genesis block is the root of the tree, and each path from the root to a node is essentially a chain. The tree will "grow": the length of each existing chain may increase, and new chains may be created, round after round. First, we define the "distance" between two chains in a tree. Intuitively, we say the distance from a "branch" chain to a "reference" chain is d if we can obtain a prefix of the reference chain by removing the last d blocks of the branch chain.

▶ **Definition 3** (Distance between two chains). *Let $\mathcal{C}$ be a chain of length ℓ, and $\mathcal{C}_1$ be a chain of length ℓ_1. We view $\mathcal{C}$ as the "reference" chain, and $\mathcal{C}_1$ to be the "branch" chain. Next, we define the distance between $\mathcal{C}$ and $\mathcal{C}_1$, and we use* $\mathsf{distance}(\textit{branch chain} \to \textit{reference chain})$*, i.e.,* $\mathsf{distance}(\mathcal{C}_1 \to \mathcal{C})$ *to denote the distance. More formally, if d is the smallest non-negative integer so that $\mathcal{C}_1[0, \ell_1 - d] \preceq \mathcal{C}$, then we say the distance between the reference chain $\mathcal{C}$ and the branch $\mathcal{C}_1$ is d, and we write* $\mathsf{distance}(\mathcal{C}_1 \to \mathcal{C}) = d$.

Now we are ready to define the distance-greedy strategies. Intuitively, a player following a distance-greedy strategy will try to extend a *set of best chains*, where the distance between the best chain and the chains in this set is quite small. Here, we consider the best chain as the branch chain and all other chains in the set of best chains as the reference chains. By the definition of the distance, we can obtain a common prefix of all reference chains by removing the last few blocks of the branch chain. Formally, we have the following definition.

▶ **Definition 4** (D-distance-greedy strategy). *Consider a blockchain protocol execution. Let P be a player of the protocol execution, and let $\mathbb{C}$ be the set of chains in player P's local view. Let $\mathcal{C}_{\text{best}}$ be the longest chain at round r, where $\ell = \mathsf{len}(\mathcal{C}_{\text{best}})$. Let D be non-negative integers. Define a set of chains $\mathbb{C}_{\text{best}}$ as*

$$\mathbb{C}_{\text{best}} = \left\{ \mathcal{C} \in \mathbb{C} \;\middle|\; \mathsf{distance}(\mathcal{C}_{\text{best}} \to \mathcal{C}) \leq D \right\}.$$

*We say P is D-*distance-greedy *if, for all r, P makes attempts to extend all chains $\mathcal{C} \in \mathbb{C}_{\text{best}}$.*

Our protocol. We present a new protocol $\Pi^\bullet$ to achieve the best possible unpredictability while only requiring a much smaller fraction (e.g., 57%) of honest stake to achieve the security properties. For simplicity, we consider the payloads in all blocks to be empty. Protocol $\Pi^\bullet$ uses a unique digital signature scheme and a hash function as building blocks.

In the blockchain initialization phase, the genesis block B_0 will be created. Given a group of PoS-players $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, a security parameter κ, and a unique digital signature scheme $(\mathsf{uKeyGen}, \mathsf{uKeyVer}, \mathsf{uSign}, \mathsf{uVerify})$, the initialization is as follows: each $P_j \in \mathcal{P}$ generates $(\textsc{sk}_j, \textsc{pk}_j) \leftarrow \mathsf{uKeyGen}(1^\kappa)$, publishes $\textsc{pk}_j$ and keeps $\textsc{sk}_j$ secret. The public keys are stored in B_0. In addition, an independent randomness $\mathsf{rand} \in \{0,1\}^\kappa$ will also be stored in B_0. That is $B_0 = \langle (\textsc{pk}_1, \textsc{pk}_2, \cdots, \textsc{pk}_n), \mathsf{rand} \rangle$. For simplicity, we assume the flat model and omit the stake distribution in the genesis block.

In the blockchain extension phase, our protocol is parameterized by $\mathsf{Context}^\bullet$, $\mathsf{Mining}^\bullet$, $\mathsf{Validate}^\bullet$, and D-$\mathsf{BestChainSet}^\bullet$. The algorithm $\mathsf{Validate}^\bullet$ takes a chain $\mathcal{C}$ (with length ℓ) and the current round number r as inputs and evaluates every block of $\mathcal{C}$. Starting from the head of $\mathcal{C}$, for every block $\mathcal{C}[i]$, where $i \in [\ell]$, the procedure $\mathsf{Validate}^\bullet$ verifies that 1) $\mathcal{C}[i]$ is linked to the previous block $\mathcal{C}[i-1]$ correctly, 2) the hash inequality is correct, and 3) the signature is correct. The algorithm D-$\mathsf{BestChainSet}^\bullet$ selects the best (longest) chain $\mathcal{C}_{\text{best}}$ and iterates through the set of chains in the local state to find all the chains in which the distances from $\mathcal{C}_{\text{best}}$ to those chains do not exceed D, and outputs the set of best chains $\mathbb{C}_{\text{best}}$. For a chain $\mathcal{C} = B_0 \| B_1 \| B_2 \| \ldots \| B_i$ in $\mathbb{C}_{\text{best}}$, some honest player P, with key pair $(\textsc{sk}, \textsc{pk})$, tries to extend $\mathcal{C}$ at round r as follows. First, P computes the context $\eta := \mathsf{Context}^\bullet(\mathcal{C})$. Here, algorithm $\mathsf{Context}^\bullet$ returns the hash value of the last block on $\mathcal{C}$, i.e., $\mathsf{Context}^\bullet(\mathcal{C}) = h(B_i)$. Then, P tries to obtain a new block using the $\mathsf{Mining}^\bullet$ algorithm. Concretely, a new block

Algorithm 1 PROTOCOL $\Pi^\bullet$.

```
State  : Initially, the set of chains $\mathbb{C}$ only consists of the genesis block. At round $r$, the
         PoS-player $P \in \mathcal{P}$, with $(\textsc{sk}, \textsc{pk})$ and local set of chains $\mathbb{C}$, proceeds as follows.
Upon receiving a chain $\mathcal{C}'$, set $\mathbb{C} := \mathbb{C} \cup \{\mathcal{C}'\}$ after verifying $\mathsf{Validate}^\bullet(\mathcal{C}', r) = 1$;
Compute $\mathbb{C}_{\text{best}} := D\text{-}\mathsf{BestChainSet}^\bullet(\mathbb{C})$;
for $\mathcal{C} \in \mathbb{C}_{\text{best}}$ do
    $\eta := \mathsf{Context}^\bullet(\mathcal{C})$; $B := \mathsf{Mining}^\bullet(\eta, r, \textsc{sk}, \textsc{pk})$;
    if $B \neq \perp$ then
        $\mathcal{C}_1 := \mathcal{C} \| B$; Broadcast $\mathcal{C}_1$;
    end
end
```

```
// Algorithms $\mathsf{Context}^\bullet$, $\mathsf{Mining}^\bullet$, $\mathsf{Validate}^\bullet$, and $D$-$\mathsf{BestChainSet}^\bullet$.
$\mathsf{Context}^\bullet(\mathcal{C})$:
    $\ell := \mathsf{len}(\mathcal{C})$; $\eta := h(\mathcal{C}[\ell])$; Return $\eta$;
$\mathsf{Mining}^\bullet(\eta, r, SK, PK)$:
    $\sigma := \mathsf{uSign}(\textsc{sk}, \langle \eta, r \rangle)$
    if $\mathsf{H}(\eta, r, \textsc{pk}, \sigma) < \mathtt{T}$ then  Create new block $B := \langle \eta, r, \textsc{pk}, \sigma \rangle$; Return B;
    else Return $\perp$
$\mathsf{Validate}^\bullet(\mathcal{C}, r)$:
    Parse $\mathcal{C}$ into $B_0 \| B_1 \| \cdots \| B_\ell$;
    for $i \in [1, \ell]$ do
        Parse $B_i$ into $\langle \eta_i, r_i, \textsc{pk}_i, \sigma_i \rangle$;
        if $h(B_{i-1}) \neq \eta_i$ or $\mathsf{H}(\eta_i, r_i, \textsc{pk}_i, \sigma_i) \geq \mathtt{T}$ or $\mathsf{uVerify}(\textsc{pk}_i, \langle \eta_i, r_i \rangle, \sigma_i) = 0$ or $r_i > r$
          then Return 0;
    end
    Return 1;
$D$-$\mathsf{BestChainSet}^\bullet$:
    Set $\mathcal{C}_{\text{best}}$ as the longest chain in $\mathbb{C}$ and $\mathbb{C}_{\text{best}} = \{\mathcal{C}_{\text{best}}\}$;
    for $\mathcal{C} \in \mathbb{C}$ do
        if $\mathsf{distance}(\mathcal{C}_{\text{best}} \to \mathcal{C}) \leq D$ then $\mathbb{C}_{\text{best}} := \mathbb{C}_{\text{best}} \cup \{\mathcal{C}\}$;
    end
```

could be returned by $\mathsf{Mining}^\bullet$ if the following hash inequality holds: $\mathsf{H}(\eta, r, \textsc{pk}, \sigma) < \mathtt{T}$, where $\sigma := \mathsf{uSign}(\textsc{sk}, \langle \eta, r \rangle)$. The new block B_{i+1} is defined as $B_{i+1} := \langle \eta, r, \textsc{pk}, \sigma \rangle$. The pseudocode of our protocol $\Pi^\bullet$ can be found in Algorithm 1.

Security analysis. The security analysis techniques outlined in [10, 14, 8, 3] can offer valuable insights for analyzing the security properties of protocols based on the single-extension design framework. However, our protocol $\Pi^\bullet$ does not adhere to this framework, requiring new analysis techniques to establish its security properties.

We can prove the security properties of protocol $\Pi^\bullet$ under the assumption of honest majority of *effective stake*. Recall that in Section 3, we obtain that the adversary can amplify its stake by a factor $e = 2.72$, so we define the effective stake of the adversary as $\beta^\bullet = 2.72\beta$. Similarly, following the D-distance-greedy strategy, honest players can amplify their stake by an amplification ratio $\hat{\mathtt{A}}^\bullet_D$, and we define $\alpha^\bullet = \hat{\mathtt{A}}^\bullet_D \cdot \alpha$. Now, we formally state the theorem.

▶ **Theorem 5.** *Consider an execution of multi-extension protocol $\Pi^\bullet$ in the random oracle model, where honest players follow the D-distance-greedy strategy while adversarial players could follow any arbitrary strategy. Additionally, all players have their stake registered at the beginning of the execution. Assume* $(\mathsf{uKeyGen}, \mathsf{uKeyVer}, \mathsf{uSign}, \mathsf{uVerify})$ *is a unique digital signature scheme, and $\alpha^\bullet = \lambda\beta^\bullet$, $\lambda > 1$. Then protocol $\Pi^\bullet$ achieves 1) chain growth, 2) common prefix, 3) chain quality, and 4) the best possible unpredictability properties.*

The proof is shown in the full version of our paper.

References

1 NXT whitepaper, 2014. URL: https://www.dropbox.com/s/cbuwrorf672c0yy/NxtWhitepaper_v122_rev4.pdf.

2 Adam Back. Hashcash – A denial of service counter-measure, 2002. URL: http://hashcash.org/papers/hashcash.pdf.

3 Vivek Bagaria, Amir Dembo, Sreeram Kannan, Sewoong Oh, David Tse, Pramod Viswanath, Xuechao Wang, and Ofer Zeitouni. Proof-of-stake longest chain protocols: Security vs predictability. *arXiv preprint*, 2019. arXiv:1910.02218.

4 Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Currencies without proof of work. In *Bitcoin Workshop*, 2016.

5 Jonah Brown-Cohen, Arvind Narayanan, Alexandros Psomas, and S Matthew Weinberg. Formal barriers to longest-chain proof-of-stake protocols. In *Proceedings of the 2019 ACM Conference on Economics and Computation*, pages 459–473, 2019. doi:10.1145/3328526.3329567.

6 Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000. doi:10.1007/s001459910006.

7 Phil Daian, Rafael Pass, and Elaine Shi. Snow White: Robustly reconfigurable consensus and applications to provably secure proof of stake. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 23–41. Springer, Heidelberg, February 2019. doi:10.1007/978-3-030-32101-7_2.

8 Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98. Springer, Heidelberg, April / May 2018. doi:10.1007/978-3-319-78375-8_3.

9 Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 139–147. Springer, Heidelberg, August 1993. doi:10.1007/3-540-48071-4_10.

10 Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, Heidelberg, April 2015. doi:10.1007/978-3-662-46803-6_10.

11 Aggelos Kiayias and Giorgos Panagiotakos. Speed-security tradeoffs in blockchain protocols. Cryptology ePrint Archive, Report 2015/1019, 2015. URL: https://eprint.iacr.org/2015/1019.

12 Jae Kwon. Tendermint: Consensus without mining, 2014. URL: https://tendermint.com/static/docs/tendermint.pdf.

13 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL: https://bitcoin.org/bitcoin.pdf.

14 Rafael Pass, Lior Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 643–673. Springer, Heidelberg, April / May 2017. doi:10.1007/978-3-319-56614-6_22.

15 Pavel Vasin. Blackcoin's proof-of-stake protocol v2, 2014. URL: http://blackcoin.org/blackcoin-pos-protocol-v2-whitepaper.pdf.

Brief Announcement: Optimal Uniform Circle Formation by Asynchronous Luminous Robots

Caterina Feletti
Department of Computer Science, Università degli Studi di Milano, Italy

Debasish Pattanayak
School of Computer Science, Carleton University, Ottawa, Canada

Gokarna Sharma
Department of Computer Science, Kent State University, OH, USA

Abstract

We study the Uniform Circle Formation (UCF) problem for a swarm of n autonomous mobile robots operating in *Look-Compute-Move* (LCM) cycles on the Euclidean plane. We assume our robots are *luminous*, i.e. equipped with a persistent light that can assume a color chosen from a fixed palette, and *opaque*, i.e. not able to see beyond a collinear robot. Robots are said to *collide* if they share positions or their paths intersect within concurrent LCM cycles. To solve UCF, a swarm of n robots must autonomously arrange themselves so that each robot occupies a vertex of the same regular n-gon not fixed in advance. In terms of efficiency, the goal is to design an algorithm that optimizes (or provides a tradeoff between) two fundamental performance metrics: *(i)* the execution time and *(ii)* the size of the color palette.

In this paper, we develop a deterministic algorithm solving UCF avoiding collisions in $O(1)$-time with $O(1)$ colors under the asynchronous scheduler, which is asymptotically optimal with respect to both time and number of colors used, the first such result. Furthermore, the algorithm proposed here minimizes for the first time what we call the *computational SEC*, i.e. the smallest circular area where robots operate throughout the whole algorithm.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Uniform Circle Formation, Robots with Lights, Autonomous Robots, Rank Encoding, Time and Color Complexities, Computational SEC

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.46

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2405.06617` [12]

Acknowledgements C. Feletti is member of the *Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica* (GNCS-INdAM).

1 Introduction

The *Look-Compute-Move* (LCM) model [13, 14] is a theoretical model used to study swarms of mobile robots and design distributed algorithms for solving collaborative problems for such systems. Robots are idle by default, but they can be activated by a scheduler. When a robot is activated, it performs an LCM cycle: it first obtains a snapshot of its surroundings (*Look*), then computes the new destination based on the snapshot (*Compute*), and finally moves straight to the computed destination (*Move*). After that, the robot becomes idle again. The scheduler can be fully synchronous ($\mathcal{FSYNC}$), semi-synchronous ($\mathcal{SSYNC}$), or asynchronous ($\mathcal{ASYNC}$). Most of the literature considers very simple and limited robots: they are assumed to be punctiform agents that can operate in the Euclidean plane, *autonomous* (no external control), *anonymous* (no internal identifiers), *indistinguishable* (no external identifiers), *homogeneous* (execute the same algorithm), and *disoriented* robots (each robot has its local coordinate system without any assumption of global orientation).

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 46; pp. 46:1–46:7
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Existing UCF deterministic solutions for $n \geq 1$ luminous-opaque robots on the plane, avoiding collisions. $x \in [1, O(\log \log n)]$.

Algorithm	Time (in epochs)	Number of Colors	Computational SEC	Scheduler
[8]	$O(1)$	$O(1)$	Not minimized	$\mathcal{FSYNC}$
[11]	$O(1)$	$O(1)$	Not minimized	$\mathcal{SSYNC}$
[11]	$O(n)$	$O(1)$	Not minimized	$\mathcal{ASYNC}$
[9]	$O(\log n)$	$O(1)$	Not minimized	
Generic [20]	$O(x)$	$O\left(n^{1/2^x}\right)$	Not minimized	
OptTime [20]	$O(1)$	$O(\sqrt{n})$	Not minimized	
OptColor [20]	$O(\log \log n)$	$O(1)$	Not minimized	
OptTime&Color (this paper)	$O(1)$	$O(1)$	Minimized	

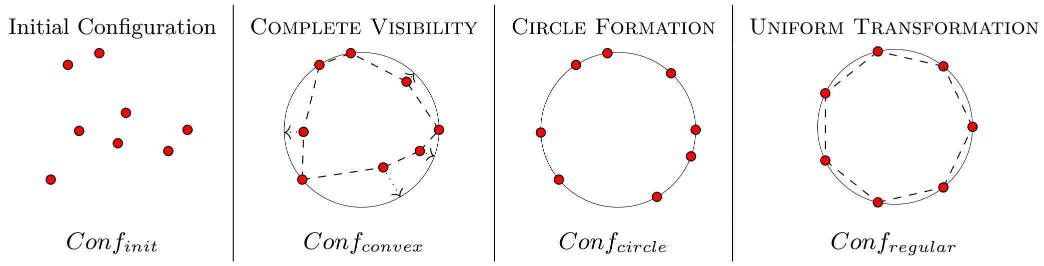
In this work, we consider *opaque* robots [1, 7, 8, 9, 20, 22, 23] thus they experience *obstructed visibility* in case of collinearities (if robots a, b, c are collinear, then a and c cannot see each other). To cope with this restrictive condition, we assume each robot is equipped with a light whose color can be updated at the beginning of its *Move* step choosing it from a fixed palette and persists until its next update. Since such a light is visible to both the robot itself and the other robots, the *luminous model* [3, 9, 16, 20, 21, 22, 23] grants robots both a persistent internal state (memory) and a direct communication means with other robots. Except for lights, robots have no other persistent memory or communication means. We say that two robots *collide* if either *(i)* they share the same position at a given time or *(ii)* their paths towards their destinations intersect within concurrent LCM cycles. We assume that our robots do not tolerate collisions and that robot movements are *rigid*, i.e., in each *Move*, the robot stops only after reaching its computed destination.

Contributions. We consider the Uniform Circle Formation (UCF) problem [4, 5, 6, 15, 17, 18, 19, 24]: starting from an arbitrary configuration where n robots lie on distinct points on a plane, robots must autonomously arrange themselves to form a regular n-gon, independently of its position, orientation, and scale. We propose a deterministic algorithm solving UCF in the luminous-opaque model under $\mathcal{ASYNC}$, avoiding collisions. Our algorithm runs in $O(1)$ time using a $O(1)$-size palette, and it minimizes a spatial metric that we call *computational SEC*, i.e. the smallest circle containing all the points the robots touch during the execution of the algorithm. Note that forcing the swarm to act within the circular area delimited by their initial configuration may represent a realistic requirement in critical scenarios (e.g. lack of space or no guarantee about the safety of the space around robots). Previous works [8, 9, 10, 11, 20] have investigated UCF under the same model: their results are summarized in Table 1 in comparison with our contribution.

Challenge and techniques. The main challenge of this work was to make robots exploit parallelism (thus achieving a $O(1)$ runtime) even in conditions of asynchrony and obstructed visibility, always keeping the size of the color palette constant and avoiding collisions among robots. For this purpose, the key techniques adopted along our algorithm include the arrangement of robots along an (inner) circle in a mirror-symmetric pattern, the Beacon Directed Curve Positioning procedure [22], and a novel *rank encoding* technique (existing techniques in [2, 9] do not fit our assumptions and requirements).

2 Algorithm Overview

Table 2 Sub-problems composing Uniform Circle Formation.



Let $Conf_{init}$ be an arbitrary initial configuration of n robots on distinct points on $\mathbb{R}^2$, all with the same color `off`. Given a configuration $Conf$, we indicate with $SEC(Conf)$ the *smallest circle enclosing* all the robots in $Conf$. Our algorithm ensures that any robot acts within the circular area delimited by $SEC(Conf_{init})$, thus minimizing the computational SEC. We now provide an overview of the different phases and procedures composing our algorithm which transforms $Conf_{init}$ into a regular configuration $Conf_{regular}$ (see Figure 1). Such procedures work in $O(1)$ time and use a $O(1)$-size palette of colors.

We factorize UCF into three sub-problems (see Table 2): *(i)* Complete Visibility, *(ii)* Circle Formation, and lastly *(iii)* Uniform Transformation. Starting from $Conf_{init}$, we *(i)* exploit the Complete Visibility solution in [22] to arrange robots on the vertices of a convex polygon, forming the configuration $Conf_{convex}$. After that, *(ii)* a simple procedure safely transforms $Conf_{convex}$ into $Conf_{circle}$ where all the robots lie on $SEC(Conf_{convex})$. From now on, let us call this circle as Cir: no robot will move out from Cir. Step *(iii)* aims to equally distribute the robots on the perimeter of Cir, thus solving UCF.

Our Uniform Transformation solution entails a different algorithmic approach according to the geometric properties of $Conf_{circle}$. Specifically, we classify $Conf_{circle}$ into three categories: $Conf_{regular}$ (where robots already form a regular polygon, so they do nothing), $Conf_{biangular}$ (biangular configuration presented in [5], where there exist two different angles α, β such that each robot forms a central angle α with one of its two adjacent robots and β with the other one), and $Conf_{periodic}$. $Conf_{biangular}$ (see Figure 2a) can be converted into a regular configuration through a similar approach to the strategy introduced in [5]: our approach guarantees robots to minimize the computational SEC. The most challenging case is the periodic configuration $Conf_{periodic}$, for which we developed a sequence of multi-step procedures to form the target regular polygon, as depicted in Figure 1.

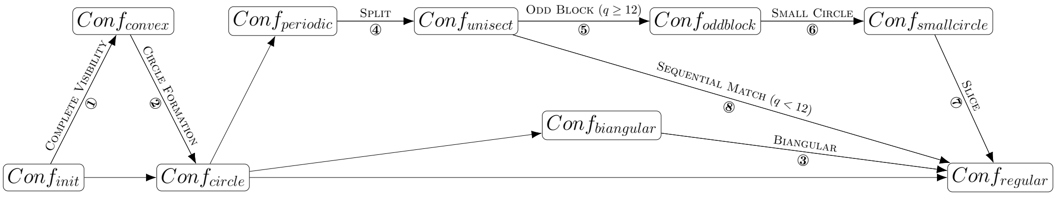
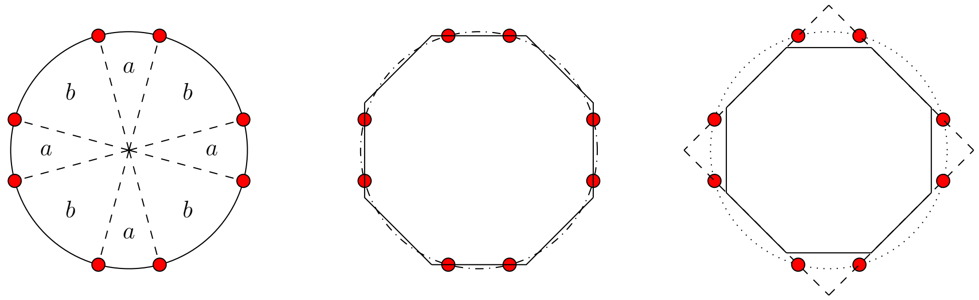


Figure 1 Transition diagram among configurations while solving UCF. The arrows without numbering denote a transition with only color change (no robot moves). The parameter q is the number of robots in each uniform sector of $Conf_{unisect}$.

3 Uniform Transformation

Biangular case

We propose a new approach to transform $Conf_{biangular}$ into $Conf_{regular}$, taking inspiration from [5]. Let P be the n-gon formed by the robots in $Conf_{biangular}$. In [5], robots spot the target regular n-gon P' which encloses P, such that robots lie on alternative edges of P', and then slide on the edges of P' until they stop on its vertices (Figure 2b). This simple approach however does not guarantee to minimize the computational SEC. Thus, we make robots spot a n-gon P'' *inscribed* in P, so that robots can slide on the larger edges of P until they reach the vertices of P'', without moving outside $SEC(Conf_{biangular})$ (Figure 2c).



(a) $Conf_{biangular}$. **(b)** Regular n-gon P' [5]. **(c)** Regular n-gon P''.

Figure 2 Arrangement of $Conf_{biangular}$ in a regular n-gon.

Periodic case

We propose a sequence of procedures to transform $Conf_{periodic}$ into $Conf_{regular}$. In $Conf_{periodic}$, all the n robots non-uniformly lie on the same circle Cir, in a periodic pattern[1], without forming a biangular configuration.

Procedure Split. $Conf_{periodic}$ is partitioned into $k \geq 2$ circular sectors $\Upsilon_0, \ldots, \Upsilon_{k-1}$ such that *(i)* they have the same arc length and *(ii)* they are size-balanced (i.e. containing the same number of robots), and *(iii)* they are chiral (i.e. the robots are arranged in an asymmetric pattern along the arc of each Υ_i). The number of sectors k depends on the degree of periodicity of $Conf_{periodic}$. We call such sectors as *uniform sectors*. Within each Υ_i, some robots will be elected as *leaders* to fix its boundaries B_i, B_{i+1}, whereas two other robots (`left`- and `right`-colored) will be elected and made to move to fix the chirality of Υ_i. Let q be the number of robots inside each Υ_i (except for its boundaries). From now on, each group of q robots works independently and in parallel within each Υ_i. The next procedures aim to uniformly arrange the q robots of each Υ_i along the arc of Υ_i, in order to cover the vertices of the target regular n-gon (also called *uniform positions*).

Procedure Sequential Match. This procedure is executed if $q < 12$, i.e., the number of robots is relatively small compared to the number of robots involved along the other procedures of the algorithm. In this case, we adopt a sequential schema to make robots reach their uniform positions along the arc of Υ_i. Specifically, following the orientation of Υ_i, robots reach their target vertex in turn.

[1] We consider an asymmetric configuration as a special case of $Conf_{periodic}$.

Procedure Odd Block. This procedure (and the following ones) is executed if $q \geq 12$. Within each sector Υ_i, two robots are elected as *guards* to fix the boundaries and chirality of a structure called *odd block*. An odd block for Υ_i is a circular sector completely contained in Υ_i, having the same origin and radius as Υ_i. Moreover, the arc of the odd block contains an odd number of uniform positions. Let $\mathcal{L}$ be the chord joining the left guard (`blockL`-colored) with the right guard (`blockR`-colored) of the odd block. One robot is elected as the *median* robot and reaches the midpoint of the arc cut by $\mathcal{L}$. The other robots on the sector arc of Υ_i are now moved to the chord $\mathcal{L}$ by implementing the Beacon Directed Curve Positioning strategy (BDCP) [22], setting their color as `chord`. See Figure 3.

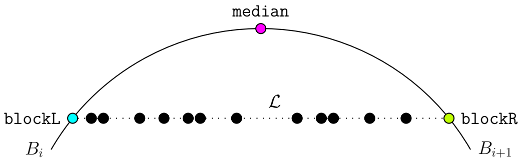


Figure 3 Odd block built inside Υ_i, delimited by the guards `blockL` and `blockR`. All the robots of the sector (except for the `median` one) have migrated to the block chord $\mathcal{L}$.

Procedure Small Circle. Let $\mathcal{C}$ be the circle within the odd block such that it passes through the median robot and such that $\mathcal{L}$ becomes its tangent. This procedure aims to place all the `chord` robots on $\mathcal{L}$ on the two halves of $\mathcal{C}$, $\mathcal{C}_w$ and $\mathcal{C}_e$, in perfect mirror-symmetry. Firstly, all the `chord` robots reach $\mathcal{C}$ traveling along the trajectories connecting their initial position on $\mathcal{L}$ with the median robot. After that, all the robots on $\mathcal{C}_w$ migrate towards $\mathcal{C}_e$ by implementing BDCP. Eventually, the robots on $\mathcal{C}_e$ split into two equal groups, and one of the groups comes back to $\mathcal{C}_w$ forming a mirror symmetric configuration on $\mathcal{C}$. See Figure 4.

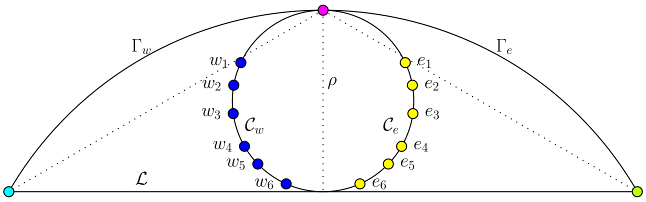


Figure 4 The small circle $\mathcal{C}$ built inside the odd block. All the robots (except for the block guards) are equally distributed on $\mathcal{C}_w$ and $\mathcal{C}_e$.

Procedure Slice. Let ρ be the diameter of $\mathcal{C}$ passing through the median robot. Let Γ_w and Γ_e be the left and right halves of the odd-block arc, cut by ρ. This procedure aims to uniformly arrange robots from $\mathcal{C}$ on the arc of the odd block. We now use a strategy to provide a rank to the robots on $\mathcal{C}_e$ ($\mathcal{C}_w$, resp.) so that a robot with rank j moves to the j-th uniform position on Γ_e (Γ_w, resp.). In particular, the robots on $\mathcal{C}_w$ move to new positions on $\mathcal{C}_w$ to encode their rank using the angular distance with a fixed robot. Thus, the robots from $\mathcal{C}_e$ can obtain their ranks using the $\mathcal{C}_w$ group as a reference. Then, the robots of each group (first the right one, then the left one) migrate on ρ on their projections, then they recompute their rank and reach their target vertices on Γ_e and Γ_w. See Figure 5.

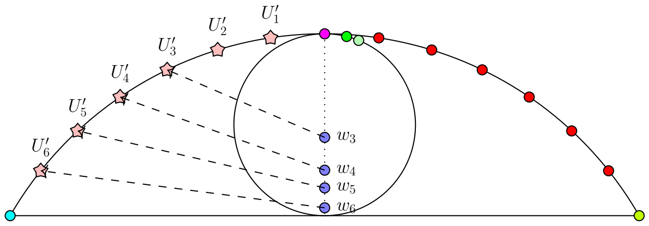


Figure 5 Each robot on ρ uses the two robots on $\mathcal{C}_e$ (here *green*) to recompute its rank j and its target uniform position U_j'.

After each uniform sector Υ_i completes the algorithm, the n robots are equally distributed on Cir, thus solving UCF. Theorem 1 summarizes our result.

▶ **Theorem 1** (Uniform Circle Formation). *Given any $Conf_{init}$ of n `off`-colored robots on distinct points on a plane, the robots reposition to $Conf_{regular}$ solving UCF in $O(1)$ epochs using $O(1)$ colors under $\mathcal{ASYNC}$, avoiding collisions, always performing within $SEC(Conf_{init})$.*

As a corollary, our UCF algorithm asymptotically optimizes both the computational time (number of epochs) and the size of the palette (number of colors), and minimizes the computational SEC.

References

1 Kaustav Bose, Manash Kumar Kundu, Ranendu Adhikary, and Buddhadeb Sau. Arbitrary pattern formation by asynchronous opaque robots with lights. *Theor. Comput. Sci.*, 849:138–158, 2021. doi:10.1016/J.TCS.2020.10.015.

2 Quentin Bramas, Anissa Lamani, and Sébastien Tixeuil. Stand up indulgent gathering. *Theoretical Computer Science*, 939:63–77, 2023. doi:10.1016/j.tcs.2022.10.015.

3 Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. Autonomous mobile robots with lights. *Theor. Comput. Sci.*, 609:171–184, 2016. doi:10.1016/J.TCS.2015.09.018.

4 Yoann Dieudonne, Ouiddad Labbani-Igbida, and Franck Petit. Circle formation of weak mobile robots. *ACM Transactions on Autonomous and Adaptive Systems*, 3, August 2006. doi:10.1145/1452001.1452006.

5 Yoann Dieudonné and Franck Petit. Swing words to make circle formation quiescent. In *SIROCCO*, volume 4474, pages 166–179, 2007. doi:10.1007/978-3-540-72951-8_14.

6 Yoann Dieudonné and Franck Petit. Squaring the circle with weak mobile robots. In *ISAAC*, pages 354–365, 2008. doi:10.1007/978-3-540-92182-0_33.

7 Caterina Feletti, Lucia Mambretti, Carlo Mereghetti, and Beatrice Palano. Computational power of opaque robots. In *3rd Symposium on Algorithmic Foundations of Dynamic Networks, SAND 2024*, volume 292, pages 13:1–13:19, 2024. doi:10.4230/LIPICS.SAND.2024.13.

8 Caterina Feletti, Carlo Mereghetti, and Beatrice Palano. Uniform circle formation for swarms of opaque robots with lights. In *20th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS*, volume 11201 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 2018. doi:10.1007/978-3-030-03232-6_21.

9 Caterina Feletti, Carlo Mereghetti, and Beatrice Palano. O$(\log n)$-time uniform circle formation for asynchronous opaque luminous robots. In *27th International Conference on Principles of Distributed Systems, OPODIS*, volume 286 of *LIPIcs*, pages 5:1–5:21, 2023. doi:10.4230/LIPICS.OPODIS.2023.5.

10 Caterina Feletti, Carlo Mereghetti, and Beatrice Palano. Uniform Circle Formation for Fully, Semi-, and Asynchronous Opaque Robots with Lights. *Applied Sciences*, 13(13):7991, 2023. doi:10.3390/app13137991.

11 Caterina Feletti, Carlo Mereghetti, Beatrice Palano, and Priscilla Raucci. Uniform circle formation for fully semi-, and asynchronous opaque robots with lights. In *23rd Italian Conference on Theoretical Computer Science, ICTCS 2022*, volume 3284, pages 207–221, 2022. URL: https://ceur-ws.org/Vol-3284/8511.pdf.

12 Caterina Feletti, Debasish Pattanayak, and Gokarna Sharma. Optimal uniform circle formation by asynchronous luminous robots. *CoRR*, abs/2405.06617, 2024. doi:10.48550/arXiv.2405.06617.

13 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. *Distributed Computing by Oblivious Mobile Robots*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012. doi:10.2200/S00440ED1V01Y201208DCT010.

14 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors. *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*. Springer, 2019. doi:10.1007/978-3-030-11072-7.

15 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta. Distributed computing by mobile robots: uniform circle formation. *Distributed Comput.*, 30(6):413–457, 2017. doi:10.1007/S00446-016-0291-X.

16 Giuseppe Antonio Di Luna, Paola Flocchini, Sruti Gan Chaudhuri, Nicola Santoro, and Giovanni Viglietta. Robots with lights: Overcoming obstructed visibility without colliding. In Pascal Felber and Vijay K. Garg, editors, *16th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2014*, volume 8756 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2014. doi:10.1007/978-3-319-11764-5_11.

17 Marcello Mamino and Giovanni Viglietta. Square formation by asynchronous oblivious robots. In *CCCG*, pages 1–6. Simon Fraser University, Vancouver, British Columbia, Canada, 2016.

18 Moumita Mondal and Sruti Gan Chaudhuri. Uniform circle formation by mobile robots. In *ICDCN*, pages 20:1–20:2. ACM, 2018. doi:10.1145/3170521.3170541.

19 Moumita Mondal and Sruti Gan Chaudhuri. Uniform circle formation by fat robots under non-uniform visibility ranges. In *ICDCN*, pages 58:1–58:5. ACM, 2020. doi:10.1145/3369740.3372779.

20 Debasish Pattanayak and Gokarna Sharma. Time-color tradeoff on uniform circle formation by asynchronous robots. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2024*, pages 987–997. IEEE, 2024. doi:10.1109/IPDPS57955.2024.00092.

21 David Peleg. Distributed coordination algorithms for mobile robot swarms: New directions and challenges. In *IWDC*, pages 1–12, 2005. doi:10.1007/11603771_1.

22 Gokarna Sharma, Ramachandran Vaidyanathan, and Jerry L. Trahan. Constant-time complete visibility for asynchronous robots with lights. In Paul G. Spirakis and Philippas Tsigas, editors, *19th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2017*, volume 10616 of *Lecture Notes in Computer Science*, pages 265–281. Springer, 2017. doi:10.1007/978-3-319-69084-1_18.

23 Gokarna Sharma, Ramachandran Vaidyanathan, Jerry L. Trahan, Costas Busch, and Suresh Rai. O(log n)-time complete visibility for asynchronous robots with lights. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017*, pages 513–522. IEEE Computer Society, 2017. doi:10.1109/IPDPS.2017.51.

24 Giovanni Viglietta. Uniform circle formation. *Chapter 5 of [14]*, pages 83–108, 2019. doi:10.1007/978-3-030-11072-7_5.

Brief Announcement: Agreement Tasks in Fault-Prone Synchronous Networks of Arbitrary Structures

Pierre Fraigniaud
Institut de Recherche en Informatique Fondamentale (IRIF)
CNRS and Université Paris Cité, France

Minh Hang Nguyen
Institut de Recherche en Informatique Fondamentale (IRIF)
CNRS and Université Paris Cité, France

Ami Paz
Interdisciplinaire des Sciences du Numérique (LISN)
CNRS and Université Paris-Saclay, France

Abstract

Consensus is arguably the most studied problem in distributed computing as a whole, and particularly in distributed message-passing settings. Research on consensus has considered various failure types, memory constraints, and much more. Surprisingly, almost all of this work assumes that messages are passed in a complete network, i.e., each process has a direct link to every other process. Set agreement, a relaxed variant of consensus, has also been heavily studied in different settings, yet research on it has also been limited to complete networks. We address this situation by considering consensus and set agreement in general networks, i.e., that can have an arbitrary graph G as their communication graph. We focus on fault-prone networks, where up to t nodes may crash and irrevocably stop communicating, and present upper and lower bounds for such networks. We establish the following collection of results:

- The consensus algorithm by [Castañeda et al., 2023] is optimal for *all* graphs, and not only for symmetric graphs.
- This algorithm can be extended to a generic algorithm for k-set agreement, for every $k \geq 1$. For $k = 1$, our generic algorithm coincides with the existing one for consensus.
- All these algorithms can be extended to the case where the number t of failures exceeds the connectivity κ of the graph, while the existing consensus algorithm assumed that $t < \kappa$.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Consensus, set-agreement, fault tolerance, crash failures

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.47

Funding ANR projects DUCAT (ANR-20-CE48-0006), QuDATA (ANR-18-CE47-0010), and TEMPORAL (ANR-22-CE48-0001); the European Union's Horizon 2020 program H2020-MSCA -COFUND-2019 Grant agreement n° 945332.
Pierre Fraigniaud: Additional support from ANR projects DUCAT (ANR-20-CE48-0006) and QuDATA (ANR-18-CE47-0010).
Minh Hang Nguyen: Additional support from ANR projects DUCAT (ANR-20-CE48-0006), TEMPORAL (ANR-22-CE48-0001), and the European Union's Horizon 2020 program H2020-MSCA -COFUND-2019 Grant agreement n° 945332.

Acknowledgements The authors want to thank Mikaël Rabie for early discussions on the subject.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 47; pp. 47:1–47:5
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The standard *synchronous t-resilient message-passing* model, for $t \geq 0$, assumes $n \geq 2$ nodes labeled from 1 to n and connected as a clique, i.e., as a complete graph K_n. Computation proceeds in synchronous rounds, during which every node can send a message to every other node, receive the message sent by every other node, and perform some local computation. Up to t nodes may crash during the execution of an algorithm, and when a node v crashes at some round $r \geq 1$ it stops functioning after round r and never recovers. Moreover, some (possibly all) of the messages sent by v at round r may be lost. This model has been extensively studied in the literature [2, 7]. In particular, it is known that consensus can be solved in $t+1$ rounds in the t-resilient model [6], and this is optimal for every $t < n-1$ as far as the worst-case complexity is concerned [1, 6]. Similarly, k-set agreement, in which the cardinality of the set of output values decided by the (correct) nodes must not exceed k, is known to be solvable in $\lfloor t/k \rfloor + 1$ rounds, which are also necessary [4].

It is only very recently that the synchronous t-resilient model has been extended to settings in which the complete communication graph K_n is replaced by an arbitrary communication graph G [3, 5]. Specifically, let $\kappa(G)$ denote the node-connectivity of G, which is the smallest number of nodes whose removal disconnects G. If the number of failures is smaller than the connectivity of the graph, i.e., if $t < \kappa(G)$, then consensus in G can be solved in $\text{radius}(G, t)$ rounds in the t-resilient model [3], where $\text{radius}(G, t)$ generalizes the standard notion of graph radius to the scenarios in which up to t nodes may crash. For $t = 0$, $\text{radius}(G, 0)$ is the standard radius of the graph G, and, for the complete graph K_n, $\text{radius}(K_n, t) = t + 1$ for every $0 \leq t < n - 1$ (while $\text{radius}(K_n, n-1) = n - 1$). Therefore, the $\text{radius}(G, t)$ upper bound for consensus in G in the t-resilient model generalizes the seminal $t + 1$ upper bound for consensus in K_n in the same model. The algorithm of [3] is *oblivious*, that is, the output of a node is solely based on the set of pairs (*node-identifier, input-value*) collected by that node during $\text{radius}(G, t)$ rounds (and not, e.g., from whom, when, and how many times it received each of these pairs). In other words, the consensus algorithm of [3] is generic, i.e., it applies to any graph G.

For a fixed graph G, the optimality of the consensus algorithm performing in $\text{radius}(G, t)$ rounds is left as an open question in [3]. It was conjectured there that for every graph G and every $0 \leq t < \kappa(G)$, no oblivious algorithms can solve consensus in G in less than $\text{radius}(G, t)$ rounds, but this was only proved for the specific case of *symmetric* (a.k.a. *vertex-transitive*) graphs. This lower bound does not come entirely as a surprise since all nodes of a symmetric graph have the same eccentricity (i.e., maximum distance to any other node), even when generalized to include crash failures. The fact that all nodes have the same eccentricity implies that they can merely be ordered according to their identifiers for selecting the output value from the received pairs (*node-identifier, input-value*). Instead, if the graph is not symmetric, a node that received a pair (*node-identifier, input-value*) after $\text{radius}(G, t)$ rounds does not known whether all the nodes have received this pair, and thus the choice of the output value from the set of received pairs is more subtle. This not only complicates the design of an upper bound but also makes the determination of a lower bound more involved.

2 The Model

We use the (synchronous) t-resilient model for networks as defined in [3]. Let $G = (V, E)$ be an n-node undirected graph, which is also connected and simple (i.e., no multiple edges nor self-loops). Initially, every node knows the graph G, that is, it knows the identifiers of all nodes and how they are connected. The uncertainty is thus not related to the initial structure of the connections, but is only due to the presence of potential failures. More specifically, computation in G proceeds as a sequence of synchronous rounds, and each node may fail by crashing – when a node crashes, it stops functioning and never recover. However, if a node v crashes at round r it may still succeeds in sending messages to a subset of its set $N(v)$ of neighbors.

For every positive integer $t \geq 0$, the t-resilient model assumes that at most t nodes may crash. A *failure pattern* is thus defined as a set $\varphi = \{(v, F_v, f_v) \mid v \in F\}$ where $F \subset V$, $0 \leq |F| \leq t$, is the set of faulty nodes in φ, and a triplet (v, F_v, f_v) designates that $v \in F$ fails at round f_v and fails to send messages to the nodes of F_v, $\varnothing \neq F_v \subseteq N(v)$, at this round. In any execution of an algorithm in the t-resilient model, the nodes know t, but do not know in advance which failure pattern occurs. The set of all failure patterns in which at most t nodes fail is denoted by $\Phi^{(t)}_{\text{all}}$.

The *eccentricity* of a node v in G with respect to a failure pattern φ, denoted by $\text{ecc}(v, \varphi)$, is the minimum number of rounds for broadcasting a message from v to all *correct* nodes of G under φ. The broadcast protocol is by flooding, i.e., when a node receives a message at round r, it forwards it to all its neighbors at round $r + 1$. Note that $\text{ecc}(v, \varphi)$ might be infinite, in case v cannot broadcast to all correct nodes in G under φ. Let

$$\Phi^{\star}_{v} = \{\varphi \in \Phi^{(t)}_{\text{all}} \mid \text{ecc}(v, \varphi) < \infty\}$$

denote the set of failure patterns in the t-resilient model in which v eventually manages to broadcast to all correct nodes. The *t-resilient radius* of G is then defined as

$$\text{radius}(G, t) = \min_{v \in V} \max_{\varphi \in \Phi^{\star}_{v}} \text{ecc}(v, \varphi).$$

Castañeda et al. [3] have designed a generic oblivious consensus algorithm which, for every graph G, and every number t of failures with $t < \kappa(G)$, runs in $\text{radius}(G, t)$ rounds. In addition, they have shown that, for every symmetric graph G, and every $t < \kappa(G)$, no oblivious algorithms can solve consensus in G with t crash failures in less than $\text{radius}(G, t)$ rounds.

3 Our Results

We extend the investigation of the t-resilient model in arbitrary graphs, in various complementary directions. The proofs of these results can be found in the full version of the paper.

3.1 Lower Bounds

We establish a general lower bound for consensus in the aforementioned synchronous t-resilient model for network, which states that the oblivious consensus algorithm from [3] is optimal among oblivious algorithms for *every* graph G, and not only for symmetric graphs.

▶ **Theorem 1.** *For every graph G and every $t < \kappa(G)$, consensus in G cannot be solved in less than* $\text{radius}(G, t)$ *rounds by an oblivious algorithm in the t-resilient model.*

3.2 Set-Agreement

We demonstrate the existence of a generic oblivious algorithm for k-set agreement. This algorithm is generic in the sense that it obeys a general structure: (1) flooding the graph with the inputs of a predetermined "core set" of nodes $C(G) \subseteq V$, for $R(G)$ rounds, and (2) after

$R(G)$ rounds, letting every node $v \in V$ pick the input of the node $u \in C(G)$ with smallest identifier among all the nodes in $C(G)$ received by v. We show that for every graph G, every $t < \kappa(G)$, and every $k \geq 1$, k-set agreement is solved in $\text{radius}(G, t, k)$ rounds, where $\text{radius}(G, t, k)$ extends the standard notion of graph radius to the case in which there are k centers, and whenever up to t nodes can crash. For $t = 0$ and $k = 1$, $\text{radius}(G, t, k)$ coincides with the standard radius of G. Moreover, for $k = 1$, $\text{radius}(G, t, 1) = \text{radius}(G, t)$.

More concretely, like in the k-center problem, we consider broadcast in G from a set $S \subseteq V$ of k nodes by flooding, and $\text{radius}(G, t, k)$ essentially denotes the minimum, taken over all sets S of k nodes, of the broadcast time of S, i.e., of the smallest number of rounds sufficient to guarantee that every non-faulty node receives information from at least one node in S. The definition is a bit more subtle though, as the broadcast time of S actually depend on the failure pattern (i.e., which nodes crash and when), and it may even be the case that S cannot broadcast at all for some failure patterns (e.g., whenever all nodes in S crash at the first round without sending any messages to their neighbors). More specifically, for every set $S \subseteq V$ of size at most k, let the eccentricity of S with respect to a failure pattern φ, denoted by $\text{ecc}(S, \varphi)$, be the minimum number of rounds such that whenever every node in S broadcasts information, every correct node of G under φ receives the information sent by at least one of the nodes in S. Let

$$\Phi_S^\infty = \{\varphi \in \Phi_{\text{all}}^{(t)} \mid \text{ecc}(S, \varphi) = \infty\},$$

and let $\Phi_S^\star = \Phi_{\text{all}}^{(t)} \smallsetminus \Phi_S^\infty$. The *$k$-center t-resilient radius* of G is then defined as

$$\text{radius}(G, t, k) = \min_{\substack{S \subseteq V \\ |S| \leq k}} \max_{\varphi \in \Phi_S^\star} \text{ecc}(S, \varphi).$$

▶ **Theorem 2.** *For every graph G, every $k \geq 1$, and every $t < \kappa(G)$, k-set agreement in G can be solved in* $\text{radius}(G, t, k)$ *rounds by an oblivious algorithm in the t-resilient model.*

3.3 Beyond the Connectivity Threshold

Finally, inspired by [5], we extend the study of consensus and set agreement in the t-resilient model in arbitrary graphs to the case where the number t of crash failures is arbitrary, i.e., not necessarily lower than the connectivity $\kappa(G)$ of the considered graph G. We show that our generic k-set agreement algorithm, which include the case of consensus for $k = 1$, can be extended to this framework, at the mere cost of relaxing consensus and k-set agreement to impose agreement to hold within each connected component of the graph resulting from removing the faulty nodes from G. Under this somehow unavoidable relaxation, we present extension of the consensus algorithm from [3] in particular, and of our k-set agreement algorithm in general, to t-resilient models for $t \geq \kappa(G)$, and express the round complexities of these algorithms in term of a straightforward extension of the radius notion to disconnected graphs.

4 Discussion

We have completed the picture for consensus in the t-resilient model for arbitrary graphs, by proving that the consensus algorithm in [3] is optimal among oblivious algorithms. Moreover, we have designed a generic (oblivious) algorithm for k-set agreement in arbitrary graph G performing in $\text{radius}(G, t, k)$ rounds under the t-resilient model, for $t < \kappa(G)$.

Our results open a vast domain for further investigations. In particular, what could be said for sets of failure patterns Φ other than $\Phi_{\text{all}}^{(t)}$? Another intriguing and potentially challenging area for further research is exploring scenarios where no upper bound on the number of failing nodes is assumed, while concentrating solely on failure patterns that do not result in the disconnection of the graph. Finally, the design of early-stopping algorithms in the t-resilient model for arbitrary graphs is also highly desirable. The algorithms in [5], early stopping and others, are very promising, but their analysis must be refined to a grain finer than the stretches of the failure patterns, by focusing, e.g., on eccentricities and radii.

References

1 Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that t-resilient consensus requires t+ 1 rounds. *Information Processing Letters*, 71(3-4):155–158, 1999. doi:10.1016/S0020-0190(99)00100-3.

2 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

3 Armando Castañeda, Pierre Fraigniaud, Ami Paz, Sergio Rajsbaum, Matthieu Roy, and Corentin Travers. Synchronous t-resilient consensus in arbitrary graphs. *Inf. Comput.*, 292:105035, 2023. doi:10.1016/J.IC.2023.105035.

4 Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. Tight bounds for k-set agreement. *J. ACM*, 47(5):912–943, 2000. doi:10.1145/355483.355489.

5 Bogdan S. Chlebus, Dariusz R. Kowalski, Jan Olkowski, and Jedrzej Olkowski. Disconnected agreement in networks prone to link failures. In *SSS*, volume 14310 of *LNCS*, pages 207–222. Springer, 2023. doi:10.1007/978-3-031-44274-2_16.

6 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983. doi:10.1137/0212045.

7 Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology.* Morgan Kaufmann, 2013.

Brief Announcement: Distinct Gathering Under Round Robin

Fabian Frei
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
Department of Computer Science, ETH Zürich, Switzerland

Koichi Wada
Hosei University, Tokyo, Japan

Abstract

We resolve one of the longest-standing questions about autonomous mobile robots in a surprising way.

Distinct Gathering is the fundamental cooperation task of letting robots, initially scattered across the plane in distinct locations, gather in an arbitrary single point. The scheduler *Round Robin* cyclically activates the robots one by one in a fixed order. When activated, a robot perceives all robot locations and moves wherever it wants based only on this information. For $n = 2$ robots, the task is trivial. What happens for $n \geq 3$ has remained an open problem for decades by now. The established conjecture declares the task to be impossible in this case. We prove that it is indeed impossible for $n = 3$ but, to great surprise, possible again for any $n \geq 4$. We go beyond the standard requirements by providing a very robust algorithm that does not require any consistency or self-consistency for the local Cartesian maps perceived by the robots and works even for non-rigid movement, that is, if robots may be unpredictably stopped and deactivated during a movement.

2012 ACM Subject Classification Computer systems organization → Robotic autonomy

Keywords and phrases Autonomous mobile robots, Distinct Gathering, Round Robin

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.48

Funding *Koichi Wada*: Supported by JSPS KAKENHI Grants 20K11685 and 21K11748.

Acknowledgements We are very grateful to all reviewers who helped us to improve this paper.

1 Introduction

Mobile robotics is an active field that studies which cooperation tasks can be performed by simplistic mobile robots. The underlying motivation is to develop cheap, reliable, and robust robots to be employed in disaster relief, for example. Suzuki and Yamashita [25] introduced the default Look-Compute-Move model: The robots are all identical, oblivious, and represented by points in the Euclidean plane. Whenever activated by some *scheduler*, a robot observes all robot locations in a local Cartesian coordinate system (which might be arbitrarily scaled, rotated, and mirrored with respect to the global coordinate system) and then moves wherever it wants. This model is sometimes referred to as OBLOT (for *oblivious robots*), and has been the object of intensive study [1, 2, 3, 4, 5, 6, 11, 12, 15, 16, 18, 23]. The simplest scheduler is the *fully synchronous* one, which always activates all robots synchronously. The second most basic and simple scheduler is arguably *Round Robin*, which lets the all robots move in orderly turns, one by one, always in the same order. The standard textbook by Flocchini et al. [13] and the newer textbook [10] provide an excellent overview of the basic models and literature.

Gathering all robots in a single point is arguably the most fundamental cooperation task and has consequently garnered the attention of many researchers [1, 2, 3, 4, 5, 6, 7, 8, 11, 14, 17, 18, 19, 20, 22, 24]. Gathering is trivial under the fully synchronous scheduler; each

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 48; pp. 48:1–48:8
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

robot moves to the center of gravity, where they meet after a single activation. This strategy fails for Round Robin; the robots will converge but never meet. Instead, the easy solution is to target a point of maximum multiplicity [25, Thm. 3.4]. Without multiplicity detection, gathering more than two robots is much more difficult. Indeed, it becomes impossible, at least if we weaken the model such to allow for initial configurations where some robots occupy the same location [21, Sec. 3]. This task variant is sometimes called *Self-Stabilizing Gathering* to distinguish it from the original task introduced in Suzuki and Yamashita's seminal paper [25], which assumes that the robots all start from distinct locations. This original version used to be called just Gathering, but is nowadays often referred to as *Distinct* Gathering to avoid ambiguity. It is an obvious and by now decades-old research problem whether Distinct Gathering is still possible for more than two robots under Round Robin, the simplest scheduler besides the fully synchronous one. Most recently, it was tackled again by an algorithm that solves the problem at least under the two additional assumptions that the robots have some internal memory and share a unit distance [26, Sect. 4, Alg. 10]. But the general consensus remained that Distinct Gathering is likely impossible under Round Robin without additional assumptions. A conjecture by Défago et al. made this explicit. [9, Conj. 2]. They corroborated their conjecture by proving the impossibility under the plausible assumption that any successful algorithm is a so-called *rapid-gathering* algorithm [9, Conj. 1]. Our results disprove both of these conjectures and resolve the open question by showing that Distinct Gathering under Round Robin, while indeed impossible for three robots, unexpectedly becomes feasible again for four and more robots. We go beyond the default robustness requirements by considering *non-rigid movement with an unknown minimum movement distance*: Any robot may be stopped at arbitrary times after some unknown minimal movement distance (e.g., due to overheating). This causes it to lose all of its memory and leaves it to wait for its next activation by the scheduler. We can provide an algorithm that works even under these exceptionally adverse conditions.

2 Detailed Model Description

In this section, we define the task to be solved, the robot model, and the considered scheduler.

▶ **Definition 1** (Gathering and Distinct Gathering). Gathering *is the task to gather n robots, which are initially scattered arbitrarily across the Euclidean plane in unknown locations. The gathering is achieved if after a finite number of activation steps all robots stay co-located in a single point forever – which point of the plane the robots choose as their gathering point is irrelevant.* Distinct gathering *is the same task with the additional guarantee that all robots occupy different locations in the initial configuration. We call a configuration with this property that no two robots share a location, a* distinct *configuration.*

▶ **Definition 2** (Robot Model). *The robots are all* identical *and* anonymous *(so they are indistinguishable and do not even have any internal ID) and all run the same algorithm. They are very simplistic and can see and move only when activated by an external scheduler. Upon activation a robot always executes a so-called Look-Compute-Move cycle:*

Look-Compute-Move Cycle. *The robot first looks around and detects all current robot locations. Then it uses this information, which is sometimes called a* snapshot*, to decide where to move and what trajectory to follow. For the upper bounds announced in this paper, we even restrict ourselves to robots moving on straight lines towards the computed target. Finally, the robot moves to its target along the pre-computed path, ignoring everything else.*

In particular, the robots are crash-resistant*; there movement is not affected in any way by them passing each other or being collocated with each other. Under the often standard assumption of* rigid *movement, the robot always reaches its target, which concludes the Look-Compute-Move cycle.*

Local Maps. *Our robots are located in the Euclidean plane and assumed to be dimensionless; they can thus be represented by points. The robots perceive the locations of the other robots with absolute precision, but only relative to a* Cartesian map *(i.e., a map with two orthogonal axes that have the same unit length) with the observing robot at the origin. Importantly, there are no further guarantees for consistency between the robots, neither for the perceived unit length nor the rotation of the axes nor the chirality. (Gathering would indeed be trivial with fully consistent maps.) Moreover, we do not even assume any self-consistency for these properties either. In other words, an adversary may arbitrarily scale, rotate, and mirror the map perceived by the robot upon each activation.*

No Multiplicity Detection. *We assume that the robots cannot detect multiplicity, that is, whether any perceived location is shared by multiple robots or not. This includes the location of the observing robot itself; it cannot sense directly whether or not it is accompanied.*

Obliviousness. *The robots are* stateless *(often referred to as* oblivious*): They possess no persistent memory and can thus only use the location information gathered upon activation to compute their movement path, and as soon as a robot stops, it forgets this information, too.*

We now define the scheduler under which Distinct Gathering is examined in this paper.

▶ **Definition 3** (Round Robin)**.** Round Robin *is the scheduler the activates the robots one by one, according to a fixed but previously unknown order, in an ever-continuing cycle covering all robots. We call any consecutive activation of all n robots a* round*. Just as the fully-synchronous scheduler, Round Robin is a natural special case of the* semi-synchronous *scheduler; that is, it always waits with the activation of a robot until all robots have stopped moving.*

Finally, we distinguish between the standard model of *rigid movement* and the assumption of the less reliable *non-rigid movement*, which models various adverse situations such as robots overheating or running out of energy during their movement.

▶ **Definition 4** (Rigid and Non-Rigid Movement)**.** *Robots with so-called* rigid movement *always reach their target when executing a Look-Compute-Move. The opposite is* non-rigid movement*, where a robot's movement may be stopped at any time by an adversary, as long as the robot has moved some minimum distance during its current cycle. This prematurely ends the robot's Look-Compute-Move cycle. For our algorithm, we assume the harshest variant of non-rigid movement where the robots have no knowledge of the minimum movement distances.*

3 Previous Results and Conjectures

Distinct Gathering is trivial under Round Robin for the case of $n = 2$ robots, often referred to as Rendezvous: It suffices for every activated robot to target to location of the other robot. The task becomes far more interesting and challenging for $n \geq 3$, however. Whether it is feasible or not has remained a prominent open question in the field for over two decades by now. It has been tackled repeatedly, most recently in a paper by Terai et al., who showed how to solve the problem at least using the quite strong additional assumptions that the robots have some internal memory, share a unit distance, and that all movements are rigid [26, Sect. 4, Alg. 10]. But the general consensus has remained that Distinct Gathering is most likely impossible under Round Robin under the standard assumptions outlined above. The following conjecture by Défago et al. makes this explicit. [9, Conj. 2].

▶ **Conjecture 5** (Cf. Défago et al. [9, Conj. 2]). *There is no algorithm solving Distinct Gathering under Round Robin for $n \geq 3$ robots under the default assumptions detailed in Section 2.*

They corroborate their assumption by proving [9, Thm. 16] the impossibility of Distinct Gathering under Round Robin under the assumption that any algorithm solving this problem is a so-called *rapid-gathering* algorithm [9, Conj. 1].

▶ **Definition 6** (Rapid-gathering algorithm, cf. Défago et al. [9, Def. 1]). *Any algorithm is a* rapid-gathering *algorithm if there is an initial configuration of n robots in distinct locations and a Round Robin activation order such that the algorithm gathers all robots within $n-1$ activations.*

▶ **Conjecture 7** (Cf. Défago et al. [9, Conj. 1]). *Every algorithm for Gathering under Round Robin is a rapid-gathering algorithm.*

They further substantiate this plausible claim by considering a series of convincing examples.

4 New Results

We are able to resolve the open question about the feasibility of Distinct Gathering under Round Robin, giving a quite surprising answer that might seem contradictory at first sight.

On the one hand, we prove with Theorem 8 that the problem is indeed impossible to solve for the case of $n = 3$. On the other hand, we show with Theorem 9 that it unexpectedly becomes feasible again for four and more robots, even though these robots have to pass through three-point configurations during the execution of their algorithm. We first formally state the result for three robots.

▶ **Theorem 8.** *Distinct Gathering under the Round Robin is impossible for $n = 3$ robots under the default model detailed in Section 2.*

Proof Sketch. Due to the space constraints, we can unfortunately not even provide a full proof sketch here since some of the details are quite subtle and require an extensive description. We restrict ourselves to mentioning here the following. The proof relies on first assuming towards contradiction that there is a gathering algorithm, then choosing an arbitrary initial configuration and a schedule, then considering the last configuration before last transition from a three-point configuration to a two-point configuration by the given algorithm under this schedule, and then considering both what could or could not have happened with this mentioned configuration as an alternative initial configuration under the different possible schedules and how the algorithm must further behave under the originally chosen schedule. ◀

We remark that Theorem 8 unifies and strengthens two separate impossibility results by Défago et al. [9, Theorems 14 and 15] at the same time for $n = 3$, extending one from (Self-Stabilizing) Gathering to *Distinct* Gathering and the other from so-called k-bounded schedulers to the heavily restricted *Round Robin* scheduler[1]. It confirms both Conjecture 5 and Conjecture 7 in the case $n = 3$. The natural assumption would now be that this result extends beyond three robots to the case of $n \geq 4$ because gathering more robots seems to only increase the difficulty of the problem. Indeed, any initial distinct configuration with four or more robots is forced to pass through a three-point configuration under Round Robin before a gathering can be achieved. Surprisingly, we can show that this argumentation is flawed by providing an algorithm that solves the problems for any given $n \geq 4$.

▶ **Theorem 9.** *Distinct Gathering under Round Robin can be solved for any given $n \geq 4$ under the default model detailed in Section 2. The result holds even for robots restricted to moving to their target in a straight line and assuming non-rigid movement with an unknown minimal movement distance.*

Theorem 9 disproves the established Conjectures 5 and 7 and resolves the long-standing open question about the feasibility of Distinct Gathering under Round Robin.

Proof Sketch. It is possible to distinguish seven main phases of the algorithm. It typically passes through these phases in the given order, but some phases may be skipped depending on the initial configuration. Moreover, in some cases it becomes necessary for the algorithm to restart the current phase or even return to previous phases. A graphic representation of the phases and the possible transitions between them is given in Figure 1.

Due to the space restrictions we can only provide a brief outline of the purpose of each phase.

The purpose of the first phase is to create a configuration whose robot locations form at most one *isosceles right triangle* or multiple isosceles right triangles with one robot that is part of all of them. The first phase ends once such a configuration is attained. The second phase has a very similar goal, namely a configuration with *exactly one* isosceles right triangle or multiple isosceles right triangles with one robot that is part of all of them. The third phase creates a configuration with exactly one isosceles right triangle. Everything up to start of the fourth phase maintains distinct locations for all the robots. In the fourth phase, all robots except for the one located at the vertex of the isosceles right triangle move to the closer one of the two endpoints of the base of this triangle. In the fifth phase, the robot at the vertex moves down along the altitude to the midpoint of the triangle's base, resulting in a *midpoint configuration*, which means that one robot is exactly at the midpoint between two other ones. In the sixth phase, all robots start targeting this midpoint. This phase ends as soon as a two-point configuration is created or a robot fails to reach its target. In the latter case, a non-midpoint three-point configuration may result, which triggers the start of the eighth phase that restores a midpoint configuration. If a four-point configuration results at the end of the sixth phase, then the seventh phase begins and ends by creating another unique isosceles right triangle, leading back to the fourth phase. The last option at the end of the sixth phase is that a two-point configuration is created, which starts the ninth phase. In this phase, all robots target the opposite observed location. This can either end in a gathering or create another three-point configuration, which means moving back to the sixth or eighth phase. One might suspect that the algorithm could be stuck in an infinite

[1] Note that the cited paper implicitly assumes neither consistency nor self-consistency in the maps of the robots, not even for the unit distance, as evidenced by the remark in the proof of Theorem 14 that all two-point configurations are indistinguishable. For the cited proofs, this assumption is insubstantial and easily removed by considering not arbitrary, but a very specific two-point configurations. In other cases, however, it might be crucial whether the perceived unit distance is self-consistent for each robot.

loop of robots swapping locations in the ninth phase. The proof that this does not happen relies an a sequence of invariants maintained through the execution of the previous phases. To prove termination of the algorithm, we need to show the termination of each phase, but also that looping back to previous phases can happen only a finite number of times. This is possible by showing that the configuration's diameter decreases substantially with every throwback to a previous phase, which lets us assume rigid movement after a finite number of phase transitions. Rigid movement prevents any transition back to the sixth, seventh, or eighth phase, and thus guarantees that the robots eventually all gather at the end of the ninth phase. ◀

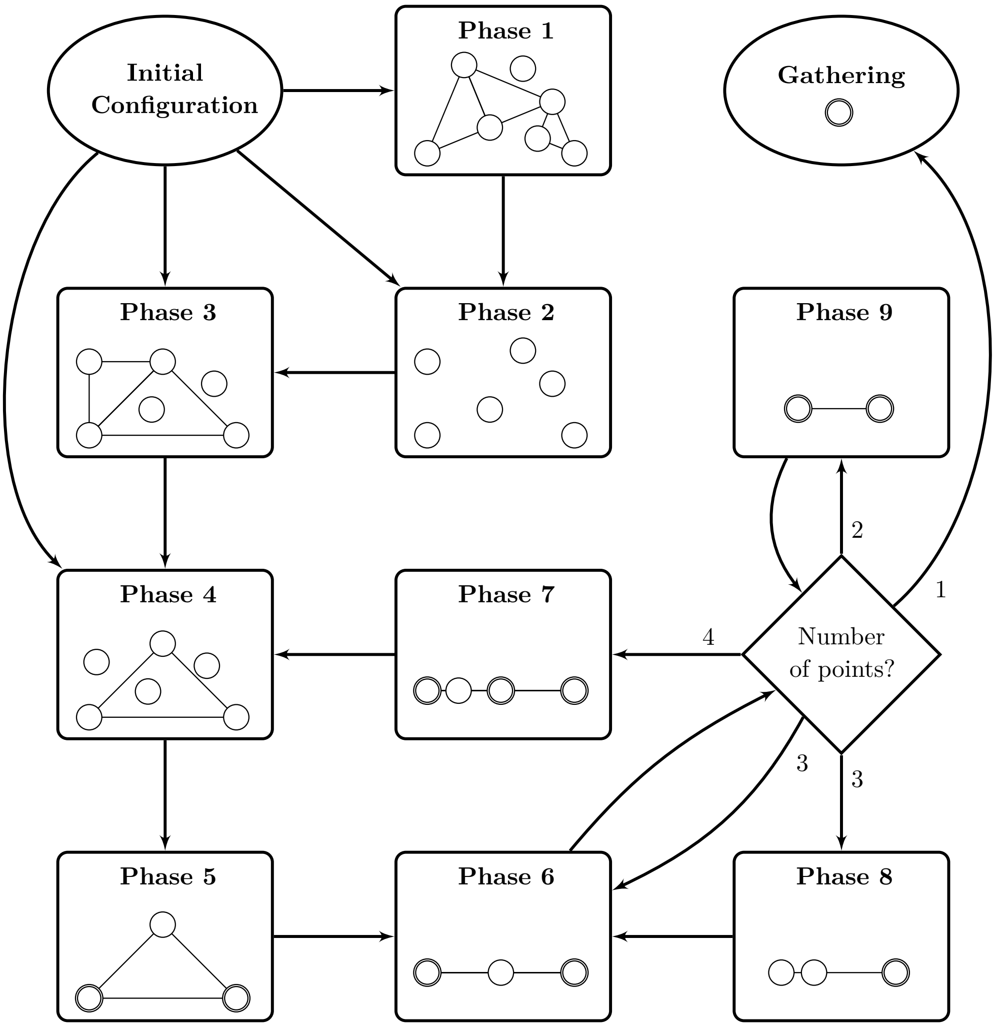


Figure 1 The phase transition diagram for the algorithm solving Distinct Gathering for more than three robots under Round Robin. Typical configurations that might occur at the start of each phase are shown. The arrows indicate the possible phase transitions. It is proved that no infinite loops can occur.

References

1 N. Agmon and D. Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. *SIAM Journal on Computing*, 36(1):56–82, 2006. doi:10.1137/050645221.

2 H. Ando, Y. Osawa, I. Suzuki, and M. Yamashita. A distributed memoryless point convergence algorithm for mobile robots with limited visivility. *IEEE Transactions on Robotics and Automation*, 15(5):818–828, 1999. doi:10.1109/70.795787.

3 Z. Bouzid, S. Das, and S. Tixeuil. Gathering of mobile robots tolerating multiple crash faults. In *the 33rd Int. Conf. on Distributed Computing Systems*, pages 334–346, 2013.

4 S. Cicerone, Di Stefano, and A. Navarra. Gathering of robots on meeting-points. *Distributed Computing*, 31(1):1–50, 2018.

5 M. Cieliebak, P. Flocchini, G. Prencipe, and N. Santoro. Distributed computing by mobile robots: Gathering. *SIAM Journal on Computing*, 41(4):829–879, 2012. doi:10.1137/100796534.

6 R. Cohen and D. Peleg. Convergence properties of the gravitational algorithms in asynchronous robot systems. *SIAM J. on Computing*, 34(15):1516–1528, 2005. doi:10.1137/S0097539704446475.

7 S. Das, P. Flocchini, G. Prencipe, N. Santoro, and M. Yamashita. Autonomous mobile robots with lights. *Theoretical Computer Science*, 609:171–184, 2016. doi:10.1016/J.TCS.2015.09.018.

8 X. Défago, M. Potop-Butucaru, and Philippe Raipin-Parvédy. Self-stabilizing gathering of mobile robots under crash or byzantine faults. *Distributed Computing*, 33(5):393–421, 2020. doi:10.1007/S00446-019-00359-X.

9 Xavier Défago, Maria Potop-Butucaru, and Philippe Raipin Parvédy. Self-stabilizing gathering of mobile robots under crash or byzantine faults. *Distributed Comput.*, 33(5):393–421, 2020. doi:10.1007/s00446-019-00359-x.

10 P. Flocchini, G. Prencipe, and N. Santoro (Eds). *Distributed Computing by Mobile Entities*. Springer, 2019.

11 P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Gathering of asynchronous robots with limited visibility. *Theoretical Computer Science*, 337(1–3):147–169, 2005. doi:10.1016/J.TCS.2005.01.001.

12 P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Arbitrary pattern formation by asynchronous oblivious robots. *Theoretical Computer Science*, 407:412–447, 2008.

13 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. *Distributed Computing by Oblivious Mobile Robots*. Synthesis Lectures on Distributed Computing Theory. Morgan and Claypool Publishers, 2012.

14 Paola Flocchini, Nicola Santoro, and Koichi Wada. On memory, communication, and synchronous schedulers when moving and computing. In *Proceeding of the 23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153 of *LIPIcs*, pages 25:1–25:17, 2019. doi:10.4230/LIPICS.OPODIS.2019.25.

15 N. Fujinaga, Y. Yamauchi, H. Ono, S. Kijima, and M. Yamashita. Pattern formation by oblivious asynchronous mobile robots. *SIAM Journal on Computing*, 44(3):740–785, 2015. doi:10.1137/140958682.

16 V. Gervasi and G. Prencipe. Coordination without communication: The case of the flocking problem. *Discrete Applied Mathematics*, 144(3):324–344, 2004. doi:10.1016/J.DAM.2003.11.010.

17 T. Izumi, Y. Katayama, N. Inuzuka, and K. Wada. Gathering autonomous mobile robots with dynamic compasses: An optimal result. In *21st DISC*, pages 298–312, 2007.

18 T. Izumi, S. Souissi, Y. Katayama, N. Inuzuka, X. Défago, K. Wada, and M. Yamashita. The gathering problem for two oblivious robots with unreliable compasses. *SIAM Journal on Computing*, 41(1):26–46, 2012. doi:10.1137/100797916.

19 J. Lin, A.S. Morse, and B.D.O. Anderson. The multi-agent rendezvous problem. parts 1 and 2. *SIAM Journal on Computing*, 46(6):2096–2147, 2007.

20 T. Okumura, K. Wada, and X. Défago. Optimal rendezvous $\mathcal{L}$-algorithms for asynchronous mobile robots with external-lights. *Theoretical Computer Science*, 979(114198), 2023.

21 Giuseppe Prencipe. On the feasibility of gathering by autonomous mobile robots. In Andrzej Pelc and Michel Raynal, editors, *Proceedings of the 12th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2005)*, volume 3499 of *Lecture Notes in Computer Science*, pages 246–261. Springer, 2005. doi:10.1007/11429647_20.

22 S. Souissi, X. Défago, and M. Yamashita. Using eventually consistent compasses to gather memory-less mobile robots with limited visibility. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1):1–27, 2009. doi:10.1145/1462187.1462196.

23 S. Souissi, T. Izumi, and K. Wada. Oracle-based flocking of mobile robots in crash-recovery model. In *Proc. 11th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 683–697, 2009.

24 I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28:1347–1363, 1999. doi:10.1137/S009753979628292X.

25 Ichiro Suzuki and Masafumi Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM J. Comput.*, 28(4):1347–1363, 1999. doi:10.1137/S009753979628292X.

26 Satoshi Terai, Koichi Wada, and Yoshiaki Katayama. Gathering problems for autonomous mobile robots with lights. *Theor. Comput. Sci.*, 941:241–261, 2023. doi:10.1016/j.tcs.2022.11.018.

Brief Announcement: Decreasing Verification Radius in Local Certification

Jan Matyáš Křišťan
Faculty of Information Technology, Czech Technical University in Prague, Czech Republic

Josef Erik Sedláček
Faculty of Information Technology, Czech Technical University in Prague, Czech Republic

Abstract

This paper deals with *local certification*, specifically locally checkable proofs: given a *graph property*, the task is to certify whether a graph satisfies the property. The verification of this certification needs to be done *locally* without the knowledge of the whole graph.

We examine the trade-off between the visibility radius and the size of certificates. We describe a procedure that decreases the radius by encoding the neighbourhood of each vertex into its certificate. We also provide a corresponding lower bound on the required certificate size increase, showing that such an approach is close to optimal.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases Local certification, locally checkable proofs, proof-labeling schemes, graphs, distributed computing

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.49

Related Version *Full Version*: `https://arxiv.org/abs/2408.10757`

Funding This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS23/205/OHK3/3T/18 and by the Czech Science Foundation Grant no. 24-12046S.

Acknowledgements We would like to thank Laurent Feuilloley for his helpful discussions and suggestions.

1 Introduction

The problem studied in this paper involves certifying a global graph property without having complete knowledge of the entire graph. In particular, we study the model of locally checkable proofs of Göös and Suomela [4].

In this model, an algorithm called a *verifier* examines the local neighbourhood of each vertex up to some fixed distance, called the *radius*. On each vertex, the verifier either accepts if it cannot deny that the graph has the desired property, or rejects if it is certain that the property does not hold. The final decision about the property is then made as follows: If the verifier rejected on at least one vertex, the decision is that the property does not hold. If it accepts on all vertices, the decision is that the property holds.

To enhance the decision-making capabilities of the model, the vertices are equipped with unique identifiers and possibly more general labels. Furthermore, each vertex is given a *certificate*. Certificates are bit-strings that are used to help the verifier in deciding the answer about the property. The verifier reads the certificates in its local view as a part of its input. For each graph that satisfies the property, the verifier must accept for at least one assignment of certificates. If the graph does not satisfy the property, the verifier must reject every assignment.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 49; pp. 49:1–49:6
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The key notion of local certification is that of a *proof labeling scheme*, which is a pair $(f, \mathcal{A})$, where $\mathcal{A}$ is the verifier and f gives each graph with the property a certificate assignment that is accepted by $\mathcal{A}$. An intuitive example is k-colorability. If k is a constant the coloring can be provided in the certificates.

Previous work and our contribution

Similar models have been studied under different names [5, 6]. The name *local certification* is a general term used for the similar models [1].

It has been previously shown how, and under which conditions, certificate size can be decreased at the cost of increasing the visibility radius [2, 3]. We provide a similar result, showing how the visibility radius can be decreased at the cost of increasing the certificate size. We also provide a corresponding lower bound on the necessary certificate size increase.

There is a crucial distinction between these two problems. While the mentioned results allow increasing the radius while decreasing the size of certificates in the general case, the implied inverse procedure of decreasing the radius works only for the very particular type of proof labeling schemes that result from the original procedure. The novelty of our results lies in allowing the decrease of the radius of *any* proof labeling schemes.

2 Preliminaries

All the graphs are assumed to be undirected and simple with possible labels. We also assume that all graphs are connected, as two different connected components have no way to interact with each other. Formally $G = (V, E, L)$ where $L\colon V \to \{0,1\}^*$. The vertices are assigned integer *identifiers*, and we assume that $V = \{1, \dots, n\}$. The set of neighbours of a vertex v is denoted as $N_G(v)$, distance between u, v as $d_G(u, v)$, and the set of vertices within distance r from v as $V[v, r]$, also called the r-local neighborhood of v.

A *graph property* is a set of graphs that is closed under isomorphism, that is, its membership does not depend on the choice of identifiers. A *certificate assignment* P for G is a function $P\colon V(G) \to \{0,1\}^*$ that associates a *certificate* with each vertex. We say that P has size s if $|P(v)| \leq s(n)$ for every v. A *verifier* is a function that takes as an input a graph G, its certificate assignment P and $v \in V(G)$ and outputs either 0 or 1.

We denote the induced subgraph $G[V[v, r]]$ as $G[v, r]$, and the restriction of P to $V[v, r]$ as $P[v, r]$, that is $P[v, r]\colon V[v, r] \to \{0,1\}^*$. A verifier $\mathcal{A}$ is *r-local* if $\mathcal{A}(G, P, v) = \mathcal{A}(G[v, r], P[v, r], v)$ for all G, P, and v. An r-local *proof labeling scheme* certifying a property of labeled graphs $\mathcal{P}$ is a pair $(f, \mathcal{A})$, where $\mathcal{A}$ is an r-local verifier and f assigns to each $G \in \mathcal{P}$ a certificate assignment such that the following properties hold.

- *Completeness*: If $G \in \mathcal{P}$, then $\mathcal{A}(G[v, r], P[v, r], v) = 1$ for all v, where $P = f(G)$.
- *Soundness*: If $G \notin \mathcal{P}$, then for every certificate assignment P', there is v such that $\mathcal{A}(G[v, r], P'[v, r], v) = 0$.

We say that $(f, \mathcal{A})$ has a size $s : \mathbb{N} \to \mathbb{N}$ if $|f(G)(v)| \leq s(|V(G)|)$ for all $G \in \mathcal{P}$ and $v \in V(G)$.

3 Decreasing the radius of a proof labeling scheme

In this section, we state that given an r-local $(f_r, \mathcal{A}_r)$ certifying a property $\mathcal{P}$, we can construct an $(r - \delta)$-local $(f, \mathcal{A})$ certifying $\mathcal{P}$ for any $\delta < r$ at the cost of increasing the certificate size. The increase of the certificate size can be expressed as a function of the size of the input graph and its maximum degree. The result is precisely formulated as follows.

▶ **Theorem 1.** *Given an r-local proof labeling scheme $(f_r, \mathcal{A}_r)$ of size s certifying a graph property $\mathcal{P}$, for every $\delta < r$, we can construct an $(r-\delta)$-local proof labeling scheme certifying $\mathcal{P}$ with certificates of the size $\mathcal{O}((\Delta-1)^{\delta}(\Delta\log(n)+s(n)+\ell(n)))$ where $\ell(n)$ is the maximum size of a label and $\Delta \geq 3$ is the maximum degree of the input graph.*

Note that in the case of $\Delta = 2$, the maximum size of a δ-neighborhood of a vertex grows only linearly with δ and we may obtain the bound on certificate size of $\mathcal{O}(\delta(\Delta\log(n)+s(n)+\ell(n)))$.

While the idea is simple, the proof is technical; therefore, due to space constraints, we decided to omit the proof from this brief announcement. The complete proof is available in the full version of the paper. Here we provide only the following overview of the proof technique.

When the verifier $\mathcal{A}_r$ is invoked on v, it is given $G[v,r]$ and $P[v,r]$ on its input. If we want to reduce that information to $G[v,r-\delta], P[v,r-\delta]$, a first step can be to *move* the now missing information into the certificates. The first obstacle comes from the fact that information in the certificates may not be true (as opposed to $G[v,r]$ provided on the input) and must be verified.

The essential idea is to have each vertex hold its δ-neighbourhood in its certificate. This allows other vertices within distance $r-\delta$ to gain information about the entire distance r neighbourhood and feed this information to the original r-local verifier.

4 Lower bound on the increase of certificate size

This section aims to show that there are proof labeling schemes for which the radius can be decreased by δ only if we also increase the certificate size by $C(\Delta-1)^{\delta-1}$, where C is a fixed constant. We present a property of labeled graphs, for which we also provide a proof labeling scheme and both an upper and a lower bound on its size.

Let $\Delta \geq 3$, then we define $\mathcal{P}_\Delta$ so that a labeled $G \in \mathcal{P}_\Delta$ if and only if it satisfies all of the following three properties. For an example of a graph with the property, see Figure 1.

Property 1 (Tree structure): G has a single vertex of degree 2, denoted as $R(G)$ (or just R), which is adjacent to two complete $(\Delta-1)$-nary trees of the same size.

Property 2 (Label structure): For every vertex v except for the root $R(G)$, the label $L(v)$ encodes an integer $a \in \{1, 2, \ldots, \Delta-1\}$ that uniquely defines its order among its siblings. Additionally, if $deg(v) = 1$, then $L(v)$ also encodes one bit $b \in \{0,1\}$. Therefore, on leaves $L(v)$ encodes a pair (a,b). The label $L(R(G))$ is empty.

This allows us to define $LT(G)$ and $RT(G)$ as the the subtrees rooted at the first and second child of $R(G)$ respectively. Furthermore, it allows us to naturally order the leaves of G. We denote as $S(v)$ the binary string created by taking the values of b on all leaves in their natural order in the subtree rooted at v. We define $S(G) = S(R(G))$.

Property 3 (String structure): $S(G) = XX$ for some binary string X, i.e. $S(G)$ is a result of concatenating a string X with itself once.

Now we describe a proof labeling scheme of $\mathcal{P}_\Delta$.

▶ **Lemma 2.** *Graph property $\mathcal{P}_\Delta$ has an r-local proof labeling scheme of size $C \cdot n/(\Delta-1)^{r-1}$ for every $r \geq 1$ and a fixed C.*

The lemma provides an upper bound on the optimal certificate size for a given radius. This is then used together with a corresponding lower bound, to show a lower bound on the necessary increase of the certificate size of $\mathcal{P}_\Delta$ when decreasing the radius.

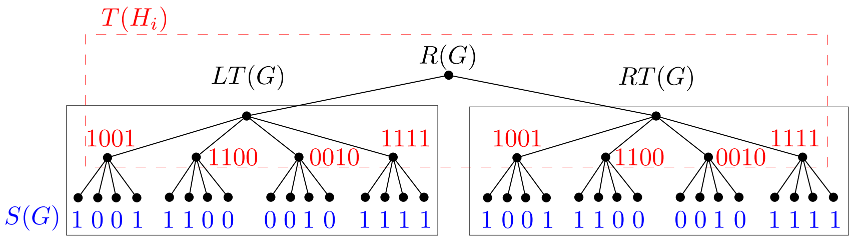


Figure 1 An example of a graph with property $\mathcal{P}_\triangle$ with $\Delta = 5$. Here, $R(G)$ is the root, $LT(G)$ and $RT(G)$ are the left and the right subtrees, $S(G)$ is the sequence in the leaves, and the red strings are certificates. The subgraph $T(H_i)$ is used in the proof of Lemma 3 and corresponds to $r = 2$.

The proof is straightforward but lengthy. The main idea is to encode for each vertex v in its certificate the string $S(v)$ of the whole subtree rooted in v. Since the verifier can see up to distance r, it is not necessary to encode the string in the vertices for which $d(v, R) < r$. See Figure 1 for an example and the full paper for the whole proof.

Now, we show a lower bound on the required certificate size to locally certify $\mathcal{P}_\triangle$.

▶ **Lemma 3.** *For all r-local proof labeling schemes certifying $\mathcal{P}_\triangle$ of size s, it holds that $s(n) \geq (n \cdot \varepsilon)/(12(\Delta - 1)^r)$ for a large enough n and all $\varepsilon < 1$.*

Proof. The idea is inspired by the proof of Theorem 6.1 of Göös and Suomela [4]. Following their approach, we will show that for every supposed proof labeling scheme of size less than $(n \cdot \varepsilon)/(12(\Delta - 1)^r)$, we can construct an instance not in $\mathcal{P}_\triangle$ which the verifier would necessarily accept.

Suppose there exists an r-local proof labeling scheme $(\mathcal{A}, f)$ certifying $\mathcal{P}_\triangle$ such that for every n' there exists $n \geq n'$ such that $s(n) < (n \cdot \varepsilon)/(12(\Delta - 1)^r)$. For an instance $H_i \in \mathcal{P}_\triangle$, let $T(H_i)$ denote $V[R(H_i), r]$. Let $\sim$ be a binary relation on $\mathcal{P}_\triangle$ defined so that $H_i \sim H_j$ if and only if $f(H_i)[T(H_i)] = f(H_j)[T(H_j)]$ and $H_i[T(H_i)] = H_j[T(H_j)]$, that is both the subgraphs on $T(H_i)$, $T(H_j)$, and their certificates as assigned by f are the same. The equality of induced subgraphs here means the equality of the identifiers, the labels, and the edges. Note that $\sim$ is an equivalence. See again Figure 1 for an illustration.

Let $\mathcal{P}_\triangle[n]$ be the set of instances in $\mathcal{P}_\triangle$ on n vertices with a fixed identifier assignment, meaning the identifier of a vertex with a given position in the tree is the same in all the instances.

▷ Claim 4. For all n', there exists $n \geq n'$ and $H_1, H_2 \in \mathcal{P}_\triangle[n]$ such that $H_1 \sim H_2$ and $S(H_1) \neq S(H_2)$.

Proof. We will show that for large enough n, the number of possible binary sequences in the leaves of instances in $\mathcal{P}_\triangle[n]$ is greater than the number of equivalence classes of $\sim$ when restricted to $\mathcal{P}_\triangle[n]$. By the assumption, each vertex has less than $(n \cdot \varepsilon)/(12(\Delta - 1)^r)$ certificate bits, thus for an instance $H_i \in \mathcal{P}_\triangle[n]$, there are at most $2^{(n\cdot\varepsilon)/(12(\Delta-1)^r)\cdot|T(H_i)|}$ different certificate assignments on $T(H_i)$, and at most $(\Delta - 1)^{|T(H_i)|}$ different assignments of labels on $T(H_i)$. The rest of the structure on $T(H_i)$, including the identifiers is fixed by the fact that $H_i \in \mathcal{P}_\triangle[n]$.

Furthermore, observe that $|T(H_i)| = 1 + 2\sum_{i=0}^{r-1}(\Delta - 1)^i \leq 3(\Delta - 1)^r$ as $\Delta \geq 3$. In total, we have that $\sim$ has on $\mathcal{P}_\triangle[n]$ at most $2^{(n\cdot\varepsilon)/4} \cdot (\Delta - 1)^{3(\Delta-1)^r}$ different classes.

On the other hand, each instance has at least $n/4$ leaves in the left subtree and thus there are at least $2^{n/4}$ different possible binary strings in the left subtree. It remains to observe that $2^{(n\cdot\varepsilon)/4} \cdot (\Delta - 1)^{3(\Delta-1)^r} < 2^{n/4}$ for large enough n. Therefore by the pigeonhole principle, there are $H_1, H_2 \in \mathcal{P}_\triangle[n]$ such that $S(H_1) \neq S(H_2)$ and $H_1 \sim H_2$. ◁

Now, we take $H_1, H_2 \in \mathcal{P}_\Delta[n]$ such that $H_1 \sim H_2$ and $S(H_1) \neq S(H_2)$ and construct $H' = (V', E', L')$ by starting with $H_1[T(H_1)] = H_2[T(H_2)]$ and completing the left subtree by $LT(H_1)$ and the right subtree by $RT(H_2)$. Formally, let $L_S(G)$ be the neighbour of $R(G)$ in $LT(G)$ and $R_S(G)$ the neighbour in $RT(G)$. Then

$$\begin{aligned} V' &= V(LT(H_1)) \cup V(RT(H_2)) \cup \{R(H_1)\} \\ E' &= E(LT(H_1)) \cup E(RT(H_2)) \cup \{R(H'), L_S(H_1)\} \cup \{R(H'), R_S(H_2)\}). \end{aligned}$$

Observe that the identifier assignment of H' is the same as those of H_1 and H_2, hence by construction, we have that H' satisfies Properties 1 and 2 and the verifier can not reject H' on their basis. Furthermore, observe that $H' \notin \mathcal{P}_\Delta$ as the string in the leaves does not satisfy Property 3.

Now, we choose the certificate assignment on H' as

$$P(v) = \begin{cases} f(H_1)(v) & \text{if } v \in LT(H') \cup \{R(H')\} \\ f(H_2)(v) & \text{otherwise} \end{cases}$$

▷ Claim 5. For all $v \in V(H')$ it holds $\mathcal{A}[H'[v, r], P[v, r]] = 1$.

It follows from the construction that the local neighbourhood of any v with is exactly the same as in the original graph. For the complete proof see the full version of the paper.

We have demonstrated that there is an instance $H' \notin \mathcal{P}_\Delta$ which is accepted by $\mathcal{A}$, contradicting the assumption that $(f, \mathcal{A})$ certifies $\mathcal{P}_\Delta$. This finishes the proof. ◀

Now, we are ready to prove there are proof labeling schemes, such that the increase of certificate size by $C(\Delta - 1)^{\delta - 1}$ is necessary when decreasing the radius by δ.

▶ **Theorem 6.** *There is an r-local proof labeling scheme of size s_r such that after decreasing its radius by δ, for any possible resulting $r - \delta$-local proof labeling scheme of size $s'_{r-\delta}$ and every large enough n, it holds that $s'_{r-\delta}(n) \geq s_r(n) \cdot C(\Delta - 1)^{\delta - 1}$ where Δ is the maximum degree of the input graph and C is a fixed constant.*

Proof. Consider the property $\mathcal{P}_\Delta$. By Lemma 2, it can be certified by a proof labeling scheme of size s_r with $s_r(n) \leq C' \cdot n/(\Delta - 1)^{r-1}$ for every large enough n. By Lemma 3, for every large enough n and a fixed C, we have:

$$s'_{r-\delta}(n) \geq (n \cdot \varepsilon)/(12(\Delta - 1)^{r-\delta}) \geq \frac{\varepsilon}{12C'} \cdot s_r(n) \cdot (\Delta - 1)^{r-1-(r-\delta)} = s_r(n) \cdot C(\Delta - 1)^{\delta - 1} ◀$$

5 Conclusion

A question to consider is the price of decreasing radius depending on the properties being certified. While our approach works in general, there may be more efficient certification methods for specific properties.

In Section 4, the presented results require that we allow labels on the vertices of the input graph. We believe that the same results can be achieved for graphs without labels, by substituting the labels with an appropriate construction.

References

1 Laurent Feuilloley. Introduction to local certification. *Discrete Mathematics & Theoretical Computer Science*, 23(Distributed Computing and Networking), 2021. doi:10.46298/dmtcs.6280.

2 Laurent Feuilloley, Pierre Fraigniaud, Juho Hirvonen, Ami Paz, and Mor Perry. Redundancy in distributed proofs. *Distributed Comput.*, 34(2):113–132, 2021. doi:10.1007/S00446-020-00386-Z.

3 Orr Fischer, Rotem Oshman, and Dana Shamir. Explicit space-time tradeoffs for proof labeling schemes in graphs with small separators. In Quentin Bramas, Vincent Gramoli, and Alessia Milani, editors, *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, volume 217 of *LIPIcs*, pages 21:1–21:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.OPODIS.2021.21.

4 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory of Computing*, 12(1):1–33, 2016. doi:10.4086/toc.2016.v012a019.

5 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Comput.*, 22(4):215–233, 2010. doi:10.1007/S00446-010-0095-3.

6 David Peleg. *Distributed computing: a locality-sensitive approach.* SIAM, 2000. doi:10.1137/1.9780898719772.

Brief Announcement: Agent-Based Leader Election, MST, and Beyond

Ajay D. Kshemkalyani
Department of Computer Science, University of Illinois at Chicago, IL, USA

Manish Kumar
Department of Computer Science & Engineering Indian Institute of Technology, Madras, India

Anisur Rahaman Molla
R. C. Bose Centre for Cryptology and Security, Indian Statistical Institute, Kolkata, India

Gokarna Sharma
Department of Computer Science, Kent State University, OH, USA

Abstract

Leader election is one of the fundamental and well-studied problems in distributed computing. In this paper, we initiate the study of leader election using mobile agents. Suppose n agents are positioned initially arbitrarily on the nodes of an arbitrary, anonymous, n-node, m-edge graph G. The agents relocate themselves autonomously on the nodes of G and elect an agent as a leader such that the leader agent knows it is a leader and the other agents know they are not leaders. The objective is to minimize time and memory requirements. Following the literature, we consider the synchronous setting in which each agent performs its operations synchronously with others and hence the time complexity can be measured in rounds. The quest in this paper is to provide solutions without agents knowing any graph parameter, such as n, a priori. We first establish that, without agents knowing any graph parameter a priori, there exists a deterministic algorithm to elect an agent as a leader in $O(m)$ rounds with $O(n \log n)$ bits at each agent. Using this leader election result, we develop a deterministic algorithm for agents to construct a minimum spanning tree of G in $O(m + n \log n)$ rounds using $O(n \log n)$ bits memory at each agent, without agents knowing any graph parameter a priori. Finally, using the same leader election result, we provide improved time/memory results for other fundamental distributed graph problems, namely, gathering, maximal independent set, and minimal dominating sets, removing the assumptions on agents knowing graph parameters a priori.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed algorithms, mobile agents, local communication, leader election, MST, MIS, gathering, minimal dominating sets, time and memory complexity, graph parameters

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.50

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2403.13716` [6]

1 Introduction

The well-studied *message-passing* distributed computing model assumes an underlying distributed network represented as an undirected graph $G = (V, E)$, where each vertex/node corresponds to a *computational device* (such as a computer or a processor), and each edge corresponds to a bi-directional communication link. Each node $v \in G$ has a distinct $\Theta(\log n)$-bit identifier, $n = |V|$. The structure of G (topology, latency) is assumed to be not known in advance, and each node typically knows only its neighboring nodes. The nodes interact with one another by sending messages (hence the name *message-passing*) to achieve a common goal. The computation proceeds according to synchronized *rounds*. In each round, each node v can perform unlimited local computation and may send a distinct message to each of its neighbors. Additionally, each node v is assumed to have no limit on storage. In the LOCAL

Table 1 Comparison of the message-passing and agent-based models.

Model	Devices	Local computation	Device storage	Neighbor communication
Message-passing	Static	Unlimited	No restriction	Messages
Agent-based	Mobile	Unlimited	Limited	Relocation

variant of this model, there is no limit on bandwidth, i.e., a node can send any size message to each of its neighbors. In the CONGEST variant, bandwidth is taken into account, i.e., a node may send only a, possibly distinct, $O(\log n)$-bit message to each of its neighbors.

In this paper, we consider the *agent-based* distributed computing model where the computational devices are modeled as *relocatable or mobile computational devices* (which we call agents). Departing from the notion of vertex/node as a *static* device in the message-passing model, the vertices/nodes serve as *containers* for the devices in the agent-based model. The agent-based model has two major differences with the message-passing model (Table 1 compares the properties of the two models).

Difference I. The graph nodes do not have identifiers, computation ability, and storage, but the devices are assumed to have distinct $O(\log n)$-bit identifiers, computation ability, and (limited) storage.

Difference II. The devices cannot send messages to other devices except the ones co-located at the same node. To send a message to a device positioned at a neighboring node, a device needs to relocate to the neighbor and can exchange information if a device is positioned at the neighbor.

Difference II is the major problem for the agent-based model. To complicate further, while a device relocates to a neighbor, the device at that neighbor might relocate to another neighbor. Therefore, the devices need to coordinate to achieve the common goal.

In this paper, we initiate the study of a graph-level task of leader election in a distributed network under the agent-based model. Leader election is one of the fundamental and well-studied problems in distributed computing due to its applications in numerous problems, such as resource allocation, reliable replication, load balancing, synchronization, membership maintenance, crash recovery, etc. Leader election can also be seen as a form of symmetry breaking, where exactly one special process or node (say a leader) is allowed to make some critical decisions. The problem of leader election in the agent-based model requires a set of agents operating in the distributed network to elect a unique leader among themselves, i.e., exactly one agent must output the decision that it is the leader.

1.1 Motivation

The agent-based model has recently found its use in multiple areas of computing. One prominent example is Martinkus *et al.* [8] which proposes AgentNet – a graph neural network (GNN) architecture, in which a collection of (neural) relocatable devices (called neural agents) *walk* the graph and collectively classify the graph-level tasks, such as triangles, cliques, and cycles. The model allows the neural agents to retrieve information from the node they are occupying, their neighboring nodes (when they visit those nodes), and the co-located devices. They showed that this agent-based model was able to detect cliques and cycles, which was shown to be impossible in the widely-studied GNN architectures based on the message-passing model (i.e., devices are static and communication is via passing messages).

Additionally, a recent study [1] has shown that the fundamental graph-level task of *triangle detection* can be solved in the agent-based model by a deterministic algorithm in $O(\Delta \log n)$ rounds with $O(\Delta \log n)$ bits at each device. In contrast, it is known that in the CONGEST message-passing model it takes $O(n^{1/3} \operatorname{polylog}(n))$ rounds to solve triangle detection by a randomized algorithm [4], which is almost tight since there is the $\Omega(n^{1/3}/\log n)$ lower bound [5, 10], and hence the agent-based model provides a clear advantage when $\Delta < n^{1/3} \operatorname{polylog}(n)$ despite restriction on communication through device relocation.

1.2 Contributions

Table 2 summarizes the problems studied and bounds obtained as well as comparison with the previous results. Specifically, we develop a deterministic algorithm for leader election with provable guarantees on two performance metrics that are fundamental to the agent-based model: *time complexity* of a solution and *storage requirement* per agent. We focus on the *deterministic* algorithms since they may be more suitable for relocatable devices. Our quest is to provide an algorithm that does not ask the agents to rely on any knowledge (neither exact nor an upper bound) on graph parameters, such as n (the network size and also the number of agents), Δ (the maximum degree of G), and D (diameter of G). This is in contrast to the message-passing model which typically assumes that n (exact n or an upper bound N on n) is known to the nodes/devices, and may be additionally Δ and D [3]. This also contrasts research in the agent-based model with known parameters (e.g., [2, 9, 11]). On the one hand, not knowing these parameters has its own merits as the solutions designed are more resilient to network changes and device faults. On the other hand, algorithm design becomes challenging since devices may not know how long to run a procedure to guarantee a solution.

Moreover, the agent-based model treats storage requirement as the first order performance metric in addition to time complexity. This is in contrast to the message-passing model where storage complexity was often neglected with the implicit assumption that the devices have no restriction on the amount of storage needed to successfully run the algorithm; in the message-passing model, the focus was given on *message complexity* (the total number of messages sent by all nodes for a solution [10]) as the first order performance metric in addition to time complexity. The goal is to use storage as small as possible (comparable to the device identifier size of $O(\log n)$ bits per device). The limited storage makes it impossible for the relocatable devices to first traverse the graph to learn the topology and then run graph computation as a second step.

Using the proposed deterministic leader election algorithm with provable guarantees on time and storage, we construct a minimum spanning tree (MST) of G, another fundamental and well-studied problem in distributed computing, for the first time in the agent-based model, without agents knowing any graph parameter a priori. We provide both time and memory complexities. Finally, as an application, using the same leader election result, we provide improved time/memory complexity algorithms for many other fundamental distributed graph problems, namely gathering, maximal independent set (MIS), and minimal dominating sets (MDS), removing the parameter assumptions in the literature.

1.3 Challenges

The message-passing model allows the nodes (processors) to send/receive messages to/from their neighbors, i.e., in a single round, a node can send a message to all its neighbors and receive messages from all its neighbors. In contrast, in the agent-based model, the messages

Table 2 Summary of previous and our results in the agent-based model. M is the memory required for the Universal Exploration Sequence (UXS) [13] and γ is the number of clusters of agents in the initial configuration. Previous results have parameter assumptions as outlined above. Our results do not have such assumptions. "$-$" means no previous result for the corresponding problem. "$\mathcal{D}$" denotes the dispersed initial configuration.

problem	previous result			our result (no parameter known)	
	time	memory/agent	known	time	memory/agent
leader	$-$	$-$	$-$	$O(m)$	$O(n\log n)$ $O(\log^2 n)$ ($\mathcal{D}$)
MST	$-$	$-$	$-$	$O(m+n\log n)$	$O(n\log n)$ $O(\log n\min\{\Delta,\log n\})$ ($\mathcal{D}$)
gathering	$O(n^3)$	$O(M+m\log n)$	n [9]	$O(m)$	$O(n\log n)$ $O(\log^2 n)$ ($\mathcal{D}$)
MIS	$O(n\Delta\log n)$	$O(\log n)$	n,Δ [11]	$O(n\Delta)$	$O(n\log n)$ $O(\log^2 n)$ ($\mathcal{D}$)
MDS	$O(\gamma\Delta\log n+n\gamma+m)$	$O(\log n)$	n,Δ,m,γ [2]	$O(m)$	$O(n\log n)$ $O(\log^2 n)$ ($\mathcal{D}$)

from an agent, if any, that are to be sent to the other agents in the neighboring nodes have to be delivered by the agent visiting those neighbors. Furthermore, it might be the case that when the agent reaches that node, the agent at that node may have already moved to another node. Therefore, any algorithm in the agent-based model needs to guarantee message delivery by synchronizing sender and receiver agents to be co-located at a node.

Additionally, the graph-level tasks (such as MST) demand each node of G to have an agent positioned on it to be able to provide a solution, i.e., if agents are not in a dispersed configuration, then MST constructed may not the MST of whole G but its sub-graph. Additionally, the MST computed may be the MST forest of graph components formed by agent positions. Notice that the initial configuration of n agents in a n-node graph G may not be dispersed.

Suppose initially the agent configuration is dispersed. Surprisingly, even in this initial configuration, the agent positioned at a node does not know this configuration. Therefore, irrespective of whether the nodes have zero, single, or multiple agents initially, it seems highly advantageous to reach a dispersed configuration.

Suppose the agents are in a dispersed configuration and the goal is to construct MST. The question is which agent starts MST construction and when. The leader election problem handles this symmetry breaking issue, since if a leader can be elected, then the authority can be given to the leader agent to initiate MST construction. The remaining agents do not participate in MST construction until the leader grants them authority to do so. Although having a leader seems to make MST construction easier and possibly other problems too, electing a leader itself turned out to be a difficult task.

2 Algorithm Overview

Initially, a graph node may have zero, one, or multiple agents. All these agents are "candidates" to become leader. A candidate needs to first become a "local leader" before becoming a "global leader". Each candidate that cannot become a "local leader" (also each "local leader" that cannot become a "global leader") will become a "non_candidate".

If an agent is initially singleton at a node, then it runs *Singleton_Election* procedure to become a local leader. If an agent is not initially non-singleton then it runs $Multiplicity\_Election$ procedure to become a local leader. After an agent becomes a local leader, it runs *Global_Election* procedure to become a global leader.

An agent r_u running *Singleton_Election* procedure at a node u will be successful in becoming a local leader if and only if all u's neighbors have initially a singleton agent positioned on them and u has the smallest degree compared to the neighboring nodes. Each initially singleton agent r_u at node u running *Singleton_Election* procedure visits the neighbors of u one by one which finishes in $2\delta_u$ rounds, where δ_u is the degree of u. If not all neighbors have initially singleton agents positioned, the agent gets to know it cannot become a local leader. It then stops executing the *Singleton_Election* procedure and becomes "non_candidate".

An agent r_u initially at node u running $Multiplicity\_Election$ procedure will be successful in becoming a local leader if and only if it has the smallest ID among the ones positioned with it initially at u. To achieve so, $Multiplicity\_Election$ procdure executes a Depth First Search (DFS) traversal ans settles the robots on each empty node visited the the traversal until there is only a singleton agent left. As soon as this condition satisfies (the smallest ID agent becomes a singleton at node w), it declares itself as a local leader[1] Except one robot, all the other robots in the $Multiplicity\_Election$ procedure become "non_candidate".

To make sure that $Multiplicity\_Election$ procedure meets the *Singleton_Election* procedure (if it is running), $Multiplicity\_Election$ procedure waits at a node for a round. *Singleton_Election* stops and the agent becomes "non_candidate" when it knows about $Multiplicity\_Election$.

After becoming a local leader (irrespective of whether through *Singleton_Election* or $Multiplicity\_Election$, the local leader agent runs *Global_Election* procedure to become a global leader. *Global_Election* procedure is again a DFS traversal as in $Multiplicity\_Election$ but with the goal to visit all the edges of G. To make it easier for other local leaders or $Multiplicity\_Election$ procedure from another agent to not mistakenly put an agent on the home node (the node where an agent becomes a local leader running *Singleton_Election* or $Multiplicity\_Election$) of a local leader the neighbor nodes are asked to store the information about a home node. The agents running $Multiplicity\_Election$ and *Global_Election* check the neighbors to confirm whether the visited empty node is in fact a home node of a local leader (or a node of an agent that is waiting to possibly become a local leader). This confirmation is obtained running $Confirm\_Empty$ procedure. If an empty node is a home node (or possible home node of an agent waiting to possibly become a local leader), $Multiplicity\_Election$ and *Global_Election* continue leaving that node empty as is. Otherwise, $Multiplicity\_Election$ puts an agent and continues, and *Global_Election* stops as it knows that $Multiplicity\_Election$ procedure from at least one agent has not finished yet.

There may be the case that while running *Global_Election*, $DFS(roundNo_i, r_i)$ of local leader r_i may *meet* $DFS(roundNo_j, r_j)$ of local leader r_j. In this case, $DFS(roundNo_i, r_i)$ continues if $roundNo_i > roundNo_j$ (if same round number, use agent IDs), otherwise $DFS(roundNo_j, r_j)$. If $DFS(roundNo_j, r_j)$ stops, then r_j becomes "non_candidate" and returns to its home node following parent pointers in $DFS(roundNo_j, r_j)$.

After a leader is elected, as an application, we use it to solve other fundamental problems. One is MST construction which was not considered in the agent-based model before. The

[1] There are cases where the parent node of w in the DFS tree built is empty and it demands the eligible robot to wait at w to decide later whether to become a local leader or a non-candidate.

rest are gathering, MIS, and MDS problems which were considered in the agent-based model before but solved assuming that the agents know one or more graph parameters a priori. We lift those assumptions and additionally provide improved time/memory bounds. This is possible by combining the leader election result with the techniques developed on the previous work under known graph parameters. The results are in Table 2.

For the MST construction, the leader plays a crucial role in synchronizing the agents. The leader ranks the agents and starts constructing an MST. It keeps its rank the highest. The leader, once its job is done, informs that second ranked agent to continue constructing MST. The second informs the third, and so on, until $(n-1)$-ranked agents pass the token to the n-th ranked. The n-th ranked agent passes the token back to the leader and one phase of MST construction finishes. It is guaranteed that at the end of this phase, there will be at least $n/2$ edges of the MST identified. Therefore, repeating this process for $O(\log n)$ phases, we have all $n-1$ edges of MST correctly identified, giving an MST of G.

2.1 Discussion on Memory Requirement

In our leader election algorithm, if n and Δ are known, a dispersed configuration can be achieved starting from any initial configuration in either $O(n\log^2 n)$ rounds using the algorithm of Sudo *et al.* [12] or in $O(m)$ rounds using the algorithm of Kshemkalyani and Sharma [7], with $O(\log n)$ bits per agent. After that, *Singleton_Election* can finish in $O(\Delta\log^2 n)$ rounds with $O(\log n)$ bits per agent. Then finally *Global_Election* procedure finishes electing a unique global leader in $O(m)$ rounds with $O(\log n)$ bits per agent. Therefore, leader election can be done with only $O(\log n)$ bits per agent (n factor improvement compared to our algorithm non-dispersed configurations). For the MST construction, a node may need to remember multiple of its neighboring edges as a part of MST and hence the total memory needed would be $O(\Delta\log n)$ bits per agent. However, notice that this memory improvement assume known n and Δ. The proposed leader election algorithm does not rely on any known graph parameters. Therefore, the proposed leader election algorithms is interesting despite $O(n\log n)$ bits memory requirement as it helped to achieve for the first time results for MST in the agent-based model and also to provide improved time/memory results for gathering, MIS, and MDS in the agent-based model, lifting the assumptions on known graph parameters.

References

1 Prabhat Kumar Chand, Apurba Das, and Anisur Rahaman Molla. Agent-based triangle counting and its applications in anonymous graphs. In *AAMAS*, 2024. doi:10.48550/arXiv.2402.03653.

2 Prabhat Kumar Chand, Anisur Rahaman Molla, and Sumathi Sivasubramaniam. Run for cover: Dominating set via mobile agents. In *ALGOWIN*, pages 133–150. Springer, 2023. doi:10.1007/978-3-031-48882-5_10.

3 Yi-Jun Chang, Seth Pettie, and Hengjie Zhang. Distributed triangle detection via expander decomposition. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 821–840. SIAM, 2019. doi:10.1137/1.9781611975482.51.

4 Yi-Jun Chang and Thatchaphol Saranurak. Improved distributed expander decomposition and nearly optimal triangle enumeration. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 66–73. ACM, 2019. doi:10.1145/3293611.3331618.

5 Taisuke Izumi and François Le Gall. Triangle finding and listing in CONGEST networks. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM*

Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017, pages 381–389. ACM, 2017. doi:10.1145/3087801.3087811.

6 Ajay D. Kshemkalyani, Manish Kumar, Anisur Rahaman Molla, and Gokarna Sharma. Agent-based MST construction. *CoRR*, abs/2403.13716, 2024. doi:10.48550/arXiv.2403.13716.

7 Ajay D. Kshemkalyani and Gokarna Sharma. Near-optimal dispersion on arbitrary anonymous graphs. In *25th International Conference on Principles of Distributed Systems, OPODIS*, pages 8:1–8:19, 2021. doi:10.4230/LIPICS.OPODIS.2021.8.

8 Karolis Martinkus, Pál András Papp, Benedikt Schesch, and Roger Wattenhofer. Agent-based graph neural networks. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL: https://openreview.net/pdf?id=8WTAh0tj2jC.

9 Anisur Rahaman Molla, Kaushik Mondal, and William K. Moses Jr. Byzantine dispersion on graphs. In *IPDPS*, pages 1–10, 2021.

10 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the distributed complexity of large-scale graph computations. In Christian Scheideler and Jeremy T. Fineman, editors, *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 405–414. ACM, 2018. doi:10.1145/3210377.3210409.

11 Debasish Pattanayak, Subhash Bhagat, Sruti Gan Chaudhuri, and Anisur Rahaman Molla. Maximal independet set via mobile agents. In *ICDCN*, pages 74–83. ACM, 2024. doi:10.1145/3631461.3631543.

12 Yuichi Sudo, Masahiro Shibata, Junya Nakamura, Yonghwan Kim, and Toshimitsu Masuzawa. Near-linear time dispersion of mobile agents, 2023. arXiv:2310.04376, doi:10.48550/arXiv.2310.04376.

13 Amnon Ta-Shma and Uri Zwick. Deterministic rendezvous, treasure hunts, and strongly universal exploration sequences. *ACM Trans. Algorithms*, 10(3):12:1–12:15, 2014. doi:10.1145/2601068.

Brief Announcement: Clock Distribution with Gradient TRIX

Christoph Lenzen
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Shreyas Srinivas[1]
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Abstract

Gradient clock synchronisation (GCS) algorithms minimise the worst-case clock offset between the nodes in a distributed network of diameter D and size n. They achieve optimal offsets of $\Theta(\log D)$ locally, i.e. between adjacent nodes [8], and $\Theta(D)$ globally [1]. A key open problem in this area is to achieve fault tolerance at minimal overhead in terms of the number of edges.

In this work, we achieve this goal under the assumption of an average-case distribution of faults, i.e., nodes fail with independent probability $p \in o(n^{-1/2})$. In more detail, we present a self-stabilising GCS algorithm for a grid-like directed graph with in- and out-degrees of 3. Note that even for tolerating a single fault, this degree is necessary. Moreover, the failure probability p is the largest possible ensuring the necessary condition that for each node at most one in-neighbour fails with probability $1 - o(1)$. Our algorithm achieves asymptotically optimal local skew of $\Theta(\log D)$ with probability $1 - o(1)$; this holds under general worst-case assumptions on link delay and clock speed variations, provided they change slowly relative to the speed of the system.

On the one hand, our results are of practical interest. As we discuss with care, the fault model is suitable for synchronously clocked hardware. Since our algorithm can simultaneously sustain a constant number of arbitrary changes due to faults in each clock cycle, it achieves sufficient robustness to dramatically increase the size of synchronously clocked Systems-on-Chip.

On the other hand, our result is of a theoretical and algorithmic nature. We show that for a worst-case distribution of f 1-local faulty nodes within our fault model's locality constraints, our algorithm achieves a local skew of $O(5^f \log D)$, while for our model with probabilistic distribution of faults the algorithm achieves $O(\log D)$. Our work raises questions for further theoretical investigation on techniques for fault tolerance and trade-offs between fault distribution and edge density of graphs.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Hardware → Fault tolerance

Keywords and phrases local skew, gradient clock synchronisation, average-case fault-tolerance, self-stabilisation, Systems-on-Chip

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.51

Related Version *Full Version*: `https://arxiv.org/abs/2301.05073` [9]

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 716562).

Acknowledgements Shreyas Srinivas is a doctoral student at the Graduate School of Computer Science, Saarbrücken.

1 The Basic Problem

The problems of distributed clock synchronisation and distribution are concerned with getting nodes in a network to agree on a common notion of time, expressed by their output logical clocks. The extent of disagreement is quantified by *clock skews* i.e. the maximum

[1] corresponding author

instantaneous difference in the output clock values of two nodes. An algorithm for this problem must seek to minimise two kinds of skew: *local skew*, i.e. clock skews between adjacent nodes, and *global skew*,i.e. clock skews between any pair of nodes in the network. Equally such an algorithm must be resilient to faults, both permanent and transient. From a pragmatic standpoint, we would like to achieve all the above properties without cluttering up our graph with replicated nodes and edges.

In this work we study the problem of distributing clock signals through a grid like *graph of diameter D* with optimal global and local clock skews of respectively $O(D)$ and $O(\log D)$, which is self stabilising and resilient to a reasonable distribution of faults. Finally, we would like to achieve the aforementioned resilience to faults by adding the minimum possible amount of edge and vertex redundancy into our network. While this last requirement arises from our desire to clock VLSI systems, the question is of independent theoretical interest since edge connectivity is an expensive resource in several domains. Our results challenge the notion that fault tolerance always requires masses of edge and vertex replication, by showing that reasonable levels of fault tolerance can be achieved at little loss of optimal performance without excessive edge replication. We summarise our desiderata below:

Fault-Tolerant Clock Synchronisation Problem (Informal)

Compute at each node of a distributed system a logical clock with the following properties.

- **Minimising Global Skew**: The skew between any pair of nodes i.e. *Global Skew* is minimised as a function of the network diameter D: $\Theta(D)$.
- **Minimising Local Skew**: The skew between adjacent pairs of nodes i.e. *Local Skew* is minimised as a function of the network diameter D: $\Theta(\log D)$.
- **Fault Tolerance**: A set of at most f permanently faulty processes according to a given fault model does not increase clock skews (up to a constant factor).
- **Self-Stabilisation**: After system-wide transient (i.e., temporary) faults, the processes re-converge to optimal skews.
- **Optimal Edge Density**: Achieve the above in a network topology with minimal node degree.

2 Our Model

We describe a slightly simplified model and leave the motivation behind it to Section 4.

- **Our Network**: We describe our *grid-like* network here. Starting with a simple connected base graph $H = (V, E)$ of minimum degree 2 and diameter D, we derive the graph $G = (V_G, E_G)$ for synchronisation as follows: for each $\ell \in \mathbb{N}$ we create a copy V_ℓ of V. Denoting by (v, ℓ) the copy of $v \in V$ in V_ℓ, we define $E_\ell := \{((v,\ell),(w,\ell+1)) \mid \{v,w\} \in E \vee v = w\}$. We now obtain G by setting $V_G := \bigcup_{\ell\in\mathbb{N}} V_\ell$ and $E_G := \bigcup_{\ell\in\mathbb{N}} E_\ell$. That is, for each *layer* $\ell \in \mathbb{N}$ we have a copy of $v \in V$, which has outgoing edges to the copies of itself and all its neighbours on layer $\ell + 1$, where ℓ is bounded from above by some value in $\Theta(\sqrt{n})$. Since V_G is a DAG, we refer to out-neighbours as *successors* and in-neighbours as *predecessors*. An example base graph and the construction of two layers from it are shown in Figure 1.

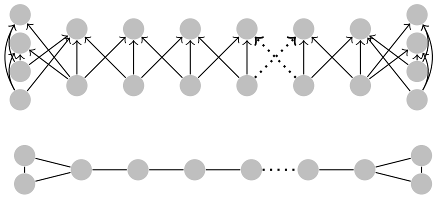


Figure 1 The figure on the bottom shows an example base graph and the figure on the top shows a two layer example of the grid graph constructed from the base graph.

- **Fault Model**: An unknown subset $F \subset V_G$ is *faulty*, meaning that these nodes do not adhere to the clock distribution protocol. We assume that each node fails independently with probability $p \in o(1/\sqrt{n})$.[2] In particular, this entails that with probability $1 - o(1)$, no node has two faulty predecessors, i.e., faults are 1-local.
- **Communication:** Each node can broadcast pulse messages on its out-edges. If node $v_\ell \in V_\ell$ broadcasts at time $t_{v,\ell}$, its successors receive its message at (potentially different) times from $[t_{v,\ell} + d - u, t_{v,\ell} + d]$. The maximum end-to-end *delay* d includes computation-induced ones. Typically, the delay *uncertainty* u is much smaller than d. We assume delays change much slower than the output clock frequency. Faulty nodes can send pulses at arbitrarily.
- **Local Clocks and Computations:** Each node (v, ℓ) has an imperfect local time reference by query access to a *hardware clock* $H_{v,\ell} \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ satisfying

  $$\forall t < t' \in \mathbb{R}_{\geq 0},\ t' - t \leq H_{v,\ell}(t') - H_{v,\ell}(t) \leq \vartheta(t' - t)$$

  for a drift parameter $\vartheta > 1$. No phase relation is assumed between the hardware clocks. Hardware clock speeds change slowly relative to the frequency of the output clocks. Computations are deterministic and can be triggered by arrival of messages or timers off the hardware clocks.
- **Output and Skew**: The algorithm outputs logical clocks in the form of pulses such the *pulses* generated by correct nodes are synchronised. For simplicity, we assume that correct nodes on layer 0 generate well-synchronised pulses at times $t^k_{v,0}$ for $k \in \mathbb{N}_{>0}$ at a frequency we control. Other correct nodes generate pulses $t^k_{v,\ell}$, $k \in \mathbb{N}_{>0}$, based on the pulse messages received from their predecessors. We seek to minimise the worst-case *local skew* that the algorithm guarantees. The local skew is defined as the largest offset between the k-th pulses of adjacent nodes on the same layer or pulses k and $k+1$ of adjacent nodes on layers ℓ and $\ell + 1$, whichever is larger. Formally, for $\ell \in \mathbb{N}$, we define

  $$\mathcal{L}_\ell := \sup_{k \in \mathbb{N}} \max_{\substack{\{v,w\} \in E \\ (v,\ell),(w,\ell) \notin F}} \{|t^k_{v,\ell} - t^k_{w,\ell}|\}, \quad \mathcal{L}_{\ell,\ell+1} := \sup_{k \in \mathbb{N}} \max_{\substack{((v,\ell),(w,\ell+1)) \in E_\ell \\ (v,\ell),(w,\ell+1) \notin F}} \{|t^{k+1}_{v,\ell} - t^k_{w,\ell+1}|\},$$

  and $\mathcal{L} := \sup_{\ell \in \mathbb{N}} \max\{\mathcal{L}_\ell, \mathcal{L}_{\ell,\ell+1}\}$.
  Between consecutive layers, we synchronise consecutive pulses. After initialisation, which is complete once the first pulse propagated through the grid, this is equivalent to a layer-dependent index shift of pulse numbers.

[2] We stress that this requirement is not stronger than that of [3, 10] and [2] for $f = 1$ in any practical sense. If faults correlate in a way clustering them together, it is likely that neighbours fail. Assuming independence (or, more generally, negative correlation) captures "faults do not cluster" in the most straightforward way that allows us to exploit this property beyond immediate neighbours.

3 The Key Technical Ingredients and Results

Our results are obtained from a medley of two lines of results, see Table 1 for a summary.

- **Clock Synchronisation and Optimal Skews**: This line dates back to the work of Fan and Lynch [4], who introduced the problem of gradient clock synchronisation (hereon *the GCS problem*), which expanded the clock synchronisation problem to general graphs. A fruitful line of work [1, 7, 8] established both lower and upper bounds of $\Theta(\log D)$ on skews that could be achieved. In fact the aptly named *GCS* (family of) *algorithms* additionally guaranteed the property of resilience to transient faults i.e. self-stabilisation. However the *GCS* algorithm is stubbornly intolerant of even a single faulty node that can lie to different neighbours which are otherwise distantly connected. In [2], the authors achieved resilience to 1-local faults by a massive replication of the vertices and edges in the original network (based on a general scheme with factor-$O(f^2)$ edge overhead), but requiring 20-fold edge replication and 4-fold vertex replication renders the scheme impractical.
- **Fault Tolerance Clock Distribution in Sparse Grids:** The other line of work [3, 10] focused on protocols for distributing a signal generated from a fault tolerant base network across a grid with fault tolerance and optimal edge connectivity. In these schemes, the nodes have no local hardware clocks of their own. They forward pulses as they received them according to the forwarding protocol. TRIX [10] has a simple pulse forwarding rule: Each node receives 3 copies of each pulse from 3 grid-adjacent in-neighbours and forwards the median copy. It achieves 1-local fault tolerance at the cost of 2 extra edges per node, but with $O(D)$ local skew.

Our Idea. It is useful to think of the pulses output by clock synchronisation and distribution schemes as discrete time points in a logical clock value they generate for each time instant. Thus we can speak of our logical clock functions being set forward or backward or have its rate of change altered. In the actual algorithm this is handled by altering the time at which successive pulses of the output clock are emitted.

We seek the best of both worlds described above. The *GCS algorithm* follows a "*move slowly to the midpoint of all your neighbours' clocks up to a discrete value* κ" rule, i.e. the *gradient rule*. Here κ is a constant picked by the algorithm designer that subsumes measurement errors from all the potential sources of uncertainty, that arise when nodes estimate their neighbours' logical clock values. The GCS algorithm offers optimal local skew, but poor fault tolerance.

In the *TRIX* distribution scheme nodes adjust their logical clocks immediately per a "*jump immediately to the median clock of three*" rule to pick one of three pulses as reference, i.e. the *median* rule. These nodes have no local reference and they merely forward pulses as they receive the second copy of each pulse. This scheme offers excellent 1-local fault tolerance but has sub-optimal $O(D)$ local skew.

Gradient TRIX. Our scheme *Gradient TRIX* attempts to combine these two rules as follows:

- It adapts a generalisation of the TRIX grid, described in Section 2. In particular, unlike TRIX, the nodes now have local clock references.
- The simple *median* pulse forwarding rule is replaced by a wait and forward rule configurable according to a parameter Λ that dictates the time period we seek to achieve for the output pulses. This fixes Λ as well as $\kappa = \Omega\left(u + \left(1 - \frac{1}{\vartheta}\right)(\Lambda - d)\right)$.

- The pulse forwarding rule is a variant of the *GCS algorithm* that safely and consistently combines the *gradient rule* with a modified *median rule.* More specifically, in addition to discrete adaptations of traditional GCS, typically called *Slow and Fast conditions* [9, Definition 9 and 10], we have a third set of *Jump Conditions* [9, Definition 11].

Intuitively, each row of the grid is playing a pass the GCS parcel game. For the duration of forwarding one pulse, each row is pretending to simulate a variation of the GCS algorithm on the base graph and then pass the baton to the next row. It is in this intuition that one can glimpse the idea behind the skew result of Theorem 1.

▶ **Theorem 1.** *If there are no faults, then $\mathcal{L}_\ell \leq 4\kappa(2 + \log D)$ for all $\ell \in \mathbb{N}$.*

This bound also accounts for suitable parameter choices that ensure that adjacent rows are closely synchronised, while a much more challenging version of the gradient property ensures synchronisation within the rows of the grid-like graph.

Up to technical details, the algorithm's self-stabilisation property is an immediate consequence of the directed propagation of pulses through the grid; once the first layer starts generating pulses at the right frequency with small local skew, the other layers follow.

▶ **Theorem 2.** *The pulse propagation algorithm can be implemented in a self-stabilising way. It stabilises within $O(\sqrt{n})$ pulses.*

Further, f permanent 1-local faults in the grid become temporary faults from the perspective of the GCS algorithm simulated on the base graph. However, for each row containing such a faulty node, the local skew might be increased by a constant factor in the worst case. Thus, we get what appears to be a substantial skew build up in the worst situation that f 1-local faults permit.

▶ **Theorem 3.** *If there are at most f faulty nodes and none in layer 0, then $\mathcal{L}_\ell \in O(5^f \kappa \log D)$.*

However, when faults are uniformly randomly distributed with each node being faulty with probability i.i.d. $o(1/\sqrt{n})$, the faults are sufficiently sparse that self-stabilisation of the simulated GCS algorithm will reduce the local skew fast enough to prevent the above exponential increase.

▶ **Theorem 4.** *With probability $1 - o(1)$, $\mathcal{L}_\ell \in O(\kappa \log D)$ for all $\ell \in \mathbb{N}$.*

A limitation of our results inherent to the directed propagation of pulses that ensures self-stabilisation of the overall scheme is that sudden changes in the timing of many links disrupt synchronisation.

▶ **Theorem 5.** *If faulty nodes do not change the timing of their output pulses, then $\mathcal{L} \in O(\kappa \log D)$ with probability $1 - o(1)$.*

On the other hand, in the considered application scenario of clock distribution on chips, the scheme is strong enough to handle the expected limited changes that occur in a clock cycle, i.e., a sub-nanosecond timescale.

▶ **Corollary 6.** *With probability $1 - o(1)$, $\mathcal{L} \in O(\kappa \log D)$ even when in each pulse (i) a constant number of faulty nodes change their output behaviour and timing, (ii) link delays vary by up to $n^{-1/2} u \log D$, and (iii) hardware clock speeds vary by up to $n^{-1/2}(\vartheta - 1)\log D$.*

[3] Given a graph topology G, the augmented graph contains a $3f + 1$-clique of replica vertices for each node v in G and $\Theta(f^2)$ copies of each edge $\{v, w\} \in G$ corresponding to all the possible pairs of the replicas of v and w

Table 1 Comparison with related work. Except GCS, "resilience" refers to Byzantine fault-tolerance, i.e., worst-case behaviour of faulty nodes. However, in our work the fault model is restricted: Only a few faulty nodes change their behaviour within a short amount of time. In turn, we are the first to simultaneously achieve optimal skew bounds, self-stabilisation, and minimal degrees.

method	global skew	local skew	resilience	self-stab.	graph topology
LW [11]	$O(1)$	$O(1)$	$< n/3$	no	complete ($D = 1$)
KL [6]	$O(1)$	$O(1)$	$< n/3$	yes	complete ($D = 1$)
HEX [3]	$O(dD)$	$d+O(u^2D/d)$	1-local	yes	grid-like, suboptimal degree
TRIX [10]	$O(uD^2)$	$O(uD)$	1-local	yes	grid-like, optimal degree
GCS [8]	$O(uD)$	$O(u \log D)$	crashes only	yes	arbitrary
Fault-tolerant GCS [2]	$O(uD)$	$O(u \log D)$	f-local	yes	$\Theta(f^2)$-augmented arbitrary graph[3]
Gradient TRIX **(this work)**	$O(uD)$	$O(u \log D)$	independent $p \in o(n^{-1/2})$	yes	grid-like, optimal degree
Gradient TRIX **(this work)**	$O(uD)$	$O(5^f u \log D)$	1-local, f faults	yes	grid-like, optimal degree

4 Motivating our Model: An Exercise in Theory Building

In this final section, we take a closer look at some of our modelling choices that might appear strange at first glance. A key motivation of this work is to produce theoretically correct algorithms which can be applied to the synchronous clocking of VLSI systems. This guides our modelling choices on two fronts:

- **Our Topology**: At a very high level, we would like to synchronise so-called clock islands on modern VLSI systems that currently rely on expensive asynchronous communication. This naturally suggests a grid-like topology. However, a simple grid does not suffice. Even with our extremely sparse network connectivity, we require each node in every row to have three neighbours in adjacent rows, meaning that the nodes at the right and left boundary "miss" a neighbour. Mathematically, the most elegant solution would be a cylinder, but embedding it on a rectangular grid induces a wasteful factor 2 overhead. Instead, we fall back to replicating each node on the right and left boundary once and connecting the two copies. Our scheme is phrased in a more general way, allowing for arbitrary base graphs of minimum degree 2. Our approach achieves this at asymptotically negligible overhead.
- **Our Fault Model**: Here, we strike a fine balance between practical viability and the theoretical optimum. A large class of permanent faults can be chalked up to manufacturing process variations and ageing. While there are correlations, the dominant contributing factors are approximately i.i.d. Further variations due to long voltage droops and temperature variations happen over times ranging a few microseconds to milliseconds [5, chp. 7]; orders of magnitude longer than the typical clock cycle. This justifies assuming that most delays do not change dramatically between consecutive pulses. Note that local oscillators are, ultimately, timed by such delays, so this applies to changes in hardware clock speeds as well.

References

1 Saâd Biaz and Jennifer Lundelius Welch. Closed Form Bounds for Clock Synchronization under Simple Uncertainty Assumptions. *Information Processing Letters*, 80:151–157, 2001. doi:10.1016/S0020-0190(01)00151-X.

2 Johannes Bund, Christoph Lenzen, and Will Rosenbaum. Fault Tolerant Gradient Clock Synchronization. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 357–365, 2019. doi:10.1145/3293611.3331637.

3 Danny Dolev, Matthias Függer, Christoph Lenzen, Martin Perner, and Ulrich Schmid. HEX: Scaling honeycombs is easier than scaling clock trees. *Journal of Computer and System Sciences*, 82(5):929–956, 2016. doi:10.1016/j.jcss.2016.03.001.

4 Rui Fan and Nancy Lynch. Gradient Clock Synchronization. In *Symposium on Principles of Distributed Computing (PODC)*, pages 320–327, 2004. doi:10.1145/1011767.1011815.

5 David Harris and N Weste. Cmos vlsi design. *ed: Pearson Education, Inc*, 2010.

6 Pankaj Khanchandani and Christoph Lenzen. Self-Stabilizing Byzantine Clock Synchronization with Optimal Precision. *Theory of Computing Systems*, 2018.

7 Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. Clock Synchronization with Bounded Global and Local Skew. In *Symposium on Foundations of Computer Science (FOCS)*, pages 509–518, 2008. doi:10.1109/FOCS.2008.10.

8 Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. Tight Bounds for Clock Synchronization. *J. ACM*, 57(2), 2010. doi:10.1145/1667053.1667057.

9 Christoph Lenzen and Shreyas Srinivas. Gradient trix, 2023. arXiv:2301.05073, doi:10.48550/arXiv.2301.05073.

10 Christoph Lenzen and Ben Wiederhake. TRIX: Low-Skew Pulse Propagation for Fault-Tolerant Hardware, 2020. arXiv:2010.01415.

11 Jennifer Lundelius Welch and Nancy A. Lynch. A New Fault-Tolerant Algorithm for Clock Synchronization. *Information and Computation*, 77(1):1–36, 1988. doi:10.1016/0890-5401(88)90043-0.

Brief Announcement: Reconfigurable Heterogeneous Quorum Systems

Xiao Li
University of California, Riverside, CA, USA

Mohsen Lesani
University of California, Santa Cruz, CA, USA

Abstract

In contrast to proof-of-work replication, Byzantine quorum systems maintain consistency across replicas with higher throughput, modest energy consumption, and deterministic liveness guarantees. If complemented with heterogeneous trust and open membership, they have the potential to serve as blockchains backbone. This paper presents a general model of heterogeneous quorum systems where each participant can declare its own quorums, and captures the consistency, availability and inclusion properties of these systems. In order to support open membership, it then presents reconfiguration protocols for heterogeneous quorum systems including joining and leaving of a process, and adding and removing of a quorum, and further, proves their correctness in the face of Byzantine attacks. The design of the protocols is informed by the trade-offs that the paper proves for the properties that reconfigurations can preserve. The paper further presents a graph characterization of heterogeneous quorum systems, and its application for reconfiguration optimization.

2012 ACM Subject Classification Computer systems organization → Reliability; Computer systems organization → Availability; Computing methodologies → Distributed algorithms

Keywords and phrases Quorum Systems, Reconfiguration, Heterogeneity

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.52

Related Version *Full Version*: `https://arxiv.org/abs/2304.02156`

1 Introduction

Banks have been traditionally closed; only established institutions could hold accounts and execute transactions. With regulations in place, this centralized model can preserve the integrity of transactions. However, it makes transactions across these institutions costly and slow; further, it keeps the power in the hands of a few. In pursuit of decentralization, Bitcoin [17] provided open membership: any node can join the Bitcoin network, and validate and process transactions. It maintains a consistent replication of an append-only ledger, called the blockchain, on a dynamic set of global hosts including potentially malicious ones. However, it suffers from a few drawbacks: low throughput, high energy consumption, and only probabilistic guarantees of commitment [9, 10].

Maintaining consistent replication in the presence of malicious processes has been the topic of Byzantine replicated systems for decades. PBFT [5] and its numerous following variants [21, 16, 22, 19, 2, 20] can maintain consistent replication when the network size is at least three times the size of potentially Byzantine coalitions, have higher throughput than Bitcoin, have modest energy consumption, give participants equal power, and provide deterministic liveness guarantees. Unfortunately, however, their quorums are uniform and their membership is closed. Their trust preferences, *i.e.*, the quorums of processes are fixed and homogeneous across the network. Further, their set of participants are fixed; thus, in contrast to proof-of-work replication that provides permissionless blockchains, classical Byzantine replication only provides permissioned blockchains.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 52; pp. 52:1–52:8
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Can the best of both worlds come together? Can we keep the consistency, throughput, modest energy consumption and equity of Byzantine replicated systems, and bring heterogeneous trust [6, 4, 1] and *open membership* to it? Openness challenges classical assumptions. With global information about the processes and their quorums, classical quorum systems could be configured at the outset to satisfy consistency and availability properties. However, open quorum systems relinquish global information as processes specify their own quorums, and can further join, leave, and reconfigure their quorums. As the other processes may be unaware of these changes, consistency and availability may be violated after and even while these reconfigurations happen.

Projects such as Ripple [18] and Stellar [15] pioneered, and follow-up research [14, 13, 8, 3] moved towards this goal, and presented quorum systems where nodes can specify their own quorums, and can join and leave. In fact, the Stellar network has a high churn. In previous works, the consistency of the network is either assumed to be maintained by user preferences or a structured hierarchy of nodes, is provided only in divided clusters of processes, or can be temporarily violated and is periodically checked across the network. Reconfigurations can compromise the consistency or availability of the replicated system. The loss of consistency can be the antecedent to a fork and double-spending. An important open problem is *reconfiguration protocols for heterogeneous quorum systems with provable security guarantees*. The protocols are expected to avoid external central oracles, or downtime.

In this paper, we first present a *general model of heterogeneous quorum systems* where each process declares its individual set of quorums, and then formally capture the properties of these systems: consistency, availability and inclusion. We then consider the *reconfiguration* of heterogeneous quorum systems: joining and leaving of a process, and adding and removing of a quorum. To cater for the protocols such as broadcast and consensus that use the quorum system, the reconfiguration protocols are expected to preserve the above properties.

The safety of consensus naturally relies on the *consistency (or quorum intersection)* property: every pair of quorums intersect at a well-behaved process. Intuitively, if an operation communicates with a quorum, and a later operation communicates with another quorum, only a well-behaved process in their intersection can make the second aware of the first. A quorum system is *available* for a process if it has a well-behaved quorum for that process. Intuitively, the quorum system is responsive to that process through that quorum. The less known property is *quorum inclusion.* Roughly speaking, every quorum should include a quorum of each of its members. This property trivially holds for homogeneous quorum systems where every quorum is uniformly a quorum of all its members, but should be explicitly maintained for heterogeneous quorum systems. We show that quorum inclusion interestingly lets processes in the included quorum make local decisions while preserving properties of the including quorum. We precisely capture and illustrate these properties.

We then present *quorum graphs*, a graph characterization of heterogeneous quorum systems with the above properties. It is known that strongly connected components of a graph form a directed acyclic graph (DAG). We prove that a quorum graph has only one *sink component*, and preserving consistency reduces to preserving quorum intersections in this component. This fact has an important implication for optimization of reconfiguration protocols. Any change outside the sink component preserves consistency, and therefore, can avoid synchronization with other processes. Thus, we present a decentralized *sink discovery protocol* that can find whether a process is in the sink.

In addition to consistency, availability and inclusion, reconfiguration protocols are expected to preserve *policies.* Each process declares its own trust policy: it specifies the quorums that it trusts. In particular, it does not trust strict subsets of its individual quorums. Thus, a policy-preserving reconfiguration should not shrink any quorum. We present a *join protocol* that preserves all the above properties. We present *trade-offs* for the properties that the leave, remove and add reconfiguration protocols can preserve. We show that there is no *leave or remove protocol* that can preserve both the policies and availability. Thus, we present two protocols: a protocol that preserves policies, and another that preserves availability. Both preserve consistency and inclusion. Then, we show that there is no *add protocol* that can preserve both the policies and consistency. Therefore, since we never sacrifice consistency, we present a protocol that preserves all properties except the policies.

We observe that under reconfiguration, *quorum inclusion is critical* to preserve not only availability but also consistency. Sometimes, reconfigurations can only eventually reconstruct inclusion, but can preserve *weaker notions of inclusion* that are sufficient to preserve consistency and availability. We capture these notions, prove that they are preserved, and use them to prove that the other properties are preserved.

In summary, this project makes the following contributions.

- A graph characterization of heterogeneous quorum systems, and its application to optimize reconfiguration and a sink discovery protocol
- Trade-offs between reconfiguration guarantees
- Reconfiguration protocols for joining and leaving of a process, and adding and removing of a quorum, and their proofs of correctness

In this short paper, we present an overview of the leave protocol. The full paper [12] presents all the above contributions more coherently.

2 Quorum Systems

Processes. A quorum system is hosted on a set of processes $\mathcal{P}$. In each execution, $\mathcal{P}$ is partitioned into *Byzantine* $\mathcal{B}$ and *well-behaved* $\mathcal{W} = \mathcal{P} \setminus \mathcal{B}$ processes. Well-behaved processes follow the given protocols; however, Byzantine processes can deviate from the protocols arbitrarily. Furthermore, a well-behaved process does not know the set of well-behaved processes $\mathcal{W}$ or Byzantine processes $\mathcal{B}$. The active processes $\mathcal{A} \subseteq \mathcal{P}$ are the current members of the system. As we will see, quorum systems can be reconfigured, and the active set can change: processes can join and the active set grows, and conversely, processes can leave, and the active set shrinks. We consider partially synchronous networks [7], *i.e.*, if both the sender and receiver are well-behaved, the message will be eventually delivered within a bounded delay after an unknown GST (Global stabilization Time). Processes can exchange messages on authenticated point-to-point links.

Individual Quorums. Processes can have different trust assumptions: trust is a subjective matter, and therefore, heterogeneous. We capture a heterogeneous model of quorum systems where each process can specify its individual set of quorums.

An *individual quorum* q of a process p is a non-empty subset of processes in $\mathcal{P}$ that p trusts to collectively perform an operation. Every quorum of a process p naturally contains p itself. (However, this is not necessary for any theorem in this paper.) By the above definition, any superset of a quorum of p is also a quorum of p. Thus, the set of quorums of p is superset-closed and has minimal members. (Consider a set of sets $S = \{\overline{s}\}$. We say that S is superset-closed, if any superset s' of any member s of S is a member of S as well.) A process p doesn't need to keep any quorum other than its minimal quorums: any of its other quorums include extra processes that p can perform operations without. Thus, we consider only the *(individual) minimal quorums* of p. Any superset of such a quorum is a *quorum* for p. We denote a set of quorums as Q. We denote the union of a set of quorums Q as $\cup Q$.

▶ **Definition 1** (Quorum System). *A heterogeneous quorum system (HQS) $\mathcal{Q}$ maps each active process to a non-empty set of individual minimal quorums.*

The mapping models the fact that each process has only a local view of its own individual minimal quorums. Further, since the behavior of Byzantine processes can be arbitrary, we leave their individual quorums unspecified.

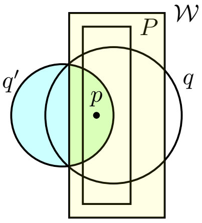


Figure 1 Quorum inclusion of q for P. Process p is a member of q that falls inside P, and q' is a quorum of p. Well-behaved processes of q' (shown as green) should be a subset of q.

The *consistency, availability and inclusion* properties are expected to be provided by a quorum system, and maintained by a reconfiguration protocol. We adapt consistency and availability for HQS [11], and define the new notion of inclusion.

▶ **Definition 2** (Consistency, Quorum Intersection). *A quorum system $\mathcal{Q}$ is consistent (i.e., has quorum intersection) at a set of well-behaved processes P iff the quorums of well-behaved processes have quorum intersection at P, i.e., $\forall p, p' \in \mathcal{W}.\ \forall q \in \mathcal{Q}(p),\ q' \in \mathcal{Q}(p').\ q \cap q' \cap P \neq \emptyset$.*

▶ **Definition 3** (Availability). *A quorum system is available for processes P at a set of well-behaved processes P' iff every process in P has at least a quorum that is a subset of P'. We say that a quorum system is available* inside *P iff it is available for P at P.*

▶ **Definition 4** (Blocking Set). *A set of processes P is a blocking set for a process p (or is p-blocking) iff P intersects every quorum of p.*

▶ **Lemma 5.** *In every quorum system that is available inside a set of processes P, every blocking set of every process in P intersects P.*

▶ **Definition 6** (Quorum inclusion). *Consider a quorum system $\mathcal{Q}$, and a subset P of its well-behaved processes. A quorum q is quorum including for P iff for every process p in the intersection of q and P, there is a quorum q' of p such that well-behaved processes of q' are a subset of q, i.e., $including(q, P) \coloneqq \forall p \in q \cap P.\ \exists q' \in \mathcal{Q}(p).\ q' \cap \mathcal{W} \subseteq q$. A quorum system $\mathcal{Q}$ is quorum including for P iff every quorum of well-behaved processes of $\mathcal{Q}$ is quorum including for P, i.e., $\forall p \in \mathcal{W}.\ \forall q \in \mathcal{Q}(p).\ including(q, P)$.*

The set P is often implicitly the set of all well-behaved processes $\mathcal{W}$. Quorum inclusion was inspired by and weakens quorum sharing [14]. Quorum sharing requires conditions on the Byzantine processes in q and q', and is too strong to maintain. We presented quorum inclusion that is weaker than quorum sharing. It requires a quorum q' only for well-behaved processes of q, and requires only the well-behaved subset of q' to be a subset of q. We will see in Section 3 that quorum inclusion is sufficient to support quorum intersection.

▶ **Definition 7** (Outlived). *A quorum system $\mathcal{Q}$ is outlived for a set of well-behaved processes $\mathcal{O}$ iff (1) $\mathcal{Q}$ is consistent at $\mathcal{O}$, (2) available inside $\mathcal{O}$, and (3) quorum including for $\mathcal{O}$.*

The safety and liveness properties of outlived processes outlive Byzantine attacks, hence the name. The protocols reconfigure an outlived quorum system into another.

3 Leave Protocol

Based on the trade-offs presented in the full paper, we present the availability-preserving and consistency-preserving protocols (AC protocols) in Algorithm 1. We then intuitively explain how it preserves the properties of the quorum system.

Algorithm 1 Leave Protocol.

```
Implements: Leave and Remove
  request : Leave
  response : LeaveComplete | LeaveFail
Variables:
  $Q$                                   ▷ Individual minimal quorums of self
  $tomb : 2^{\mathcal{P}} \leftarrow \emptyset$
  $(in\text{-}sink : \text{Boolean}, F : 2^{\mathcal{P}}) \leftarrow Discovery(Q)$
Uses:
  $tob$ : TotalOrderBroadcast
  $apl : (\cup Q) \cup F \mapsto$ AuthPPoint2PointLink
upon request Leave
    if in-sink then
        if $\forall q_1, q_2 \in Q, (q_1 \cap q_2) \setminus \{\text{self}\}$ is self-blocking then
            tob request Check(self, $Q$)
        else
            response LeaveFail
    else
        response LeaveComplete
        $apl(p)$ request Left(self)  for each $p \in F$
upon response tob, Check($p'$, $Q'$)
    if $\exists q_1, q_2 \in Q'.\ (q_1 \cap q_2) \setminus (\{p'\} \cup tomb)$ is not $p'$-blocking then
        if $p' =$ self then
            response LeaveFail
    else
        $tomb \leftarrow tomb \cup \{p'\}$
        if $p' =$ self then
            response LeaveComplete
            $apl(p)$ request Left(self)  for each $p \in F$
upon response $apl(p)$, Left($p$)
    $Q \leftarrow \{q \setminus \{p\} \mid q \in Q\}$
```

Variables and sub-protocols. Each process keeps its own set of individual minimal quorums Q. It also keeps the set *tomb* that records the processes that might have left. The full paper presents an optimization opportunity for the coordination needed to preserve consistency: when the quorum system has quorum sharing, only processes in the sink component need coordination. Therefore, each process stores whether it is in the sink component as the *in-sink* boolean, and its follower processes (*i.e.*, processes that have this process in their quorums) as the set F. (The sink information is just used for an optimization, and the protocol can execute without it.)

The protocol uses a total-order broadcast *tob*, and authenticated point-to-point links *apl* (to processes in the quorums Q and followers F). Total-order broadcast provides a broadcast interface on top of consensus [15, 14, 8, 11]. The consensus and total-order broadcast abstractions [11] require quorum intersection for safety, and quorum availability and inclusion for liveness. As we will show, the reconfiguration protocols preserve both of these properties for outlived quorum systems. We note that if a protocol naively uses *tob* to globally order and process reconfigurations, then since each process only knows its own quorums, it cannot independently check if the properties of the quorum system are preserved.

Protocol. When a process requests to leave (at L. 11), it first checks whether it is in the sink component (at L. 12). If it is not in the sink, then it can apply the optimizations that are shown with the blue color. The process can simply leave without synchronization (at L. 18); it only needs to inform its follower set so that they can preserve their quorum availability. It sends a *Left* message to its followers (at L. 19). Every well-behaved process that receives the message (at L. 29) removes the sender from its quorums (at L. 30). If the quorum system does not have quorum sharing or the sink information is not available, the protocol can be conservative (remove the blue lines) and always perform the coordination that we will consider next.

On the other hand, when the requesting process is in the sink component, its absence can put quorum intersection in danger. Therefore, it first locally checks a condition (at L. 13). The check is just an optimization not to attempt leave requests that are locally known to fail. We will consider this condition in the next subsection. If the check fails, the leave request fails (at L. 16). If the local check passes, the process broadcasts a *Check* request together with its quorums (at L. 14). If processes receive and check concurrent leave requests in different orders, they may concurrently approve leave requests for all processes in a quorum intersection. Therefore, a total-order broadcast *tob* is used to enforce a total order for processing of *Check* messages. When a process receives a *Check* request with a set of quorums Q, it locally checks a condition for Q (at L. 21). This check is similar to the check above but is repeated in the total order of deliveries by the *tob*. If the condition fails, the leave request fails (at L. 23). If it passes, the leaving process is added to the *tomb* set (at L. 25), and the leaving process informs its followers, and leaves (at lines 27 and 28). Let's now consider the condition and see how it preserves quorum intersection and inclusion.

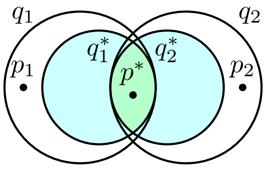


Figure 2 The Leave Protocol, Preserving Quorum Intersection.

Quorum Intersection. Let us first see an intuitive explanation of the condition, and why it preserves quorum intersection. We assume that the quorum system is outlived: there is a set of processes $\mathcal{O}$ such that the quorum system has quorum intersection at $\mathcal{O}$, quorum inclusion for $\mathcal{O}$, and quorum availability inside $\mathcal{O}$. As shown in Figure 2, consider well-behaved processes p_1 and p_2 with quorums q_1 and q_2 respectively, and let p^* be a process at the intersection of q_1 and q_2 in $\mathcal{O}$. The goal is to allow p^* to leave only if the intersection of q_1 and q_2 contains another process in $\mathcal{O}$. By the quorum inclusion property, p^* should have quorums q_1^* and q_2^* such that their well-behaved processes are included inside q_1 and q_2 respectively. Each process adds to its *tomb* set every process whose *Check* request passes. The total-order-broadcast *tob* delivers the *Check* requests in the same order across processes. Therefore, the result of the check and the updated *tomb* set is the same across processes after processing each request. Consider a *Check* request of a process p' which is ordered before that of p^*. If the check for p' is passed and it leaves, then the *tomb* set of p^* contains p'. Consider when the *Check* request of p^* is processed. The check ensures that p^* is approved to leave only if the intersection of q_1^* and q_2^* modulo the *tomb* set and p^* is p^*-blocking. By Lemma 5, since the quorum system is available inside $\mathcal{O}$, this means that the intersection of q_1^* and q_2^* after both p' and p^* leave still intersects $\mathcal{O}$. A process p in $\mathcal{O}$ remains in the intersection of q_1^* and q_2^*. Therefore, by quorum inclusion, p remains in the intersection of q_1 and q_2. Thus, outlived quorum intersection is preserved for q_1 and q_2.

Once the *tob* delivers the *Check* message of the leaving process p^* to p^* itself, it can locally decide whether it is safe to leave. We note that the local check ensures a global property: quorum intersection for the whole quorum system. We also note that both quorum inclusion and quorum availability are needed to preserve quorum intersection. Further, we note that outlived quorum intersection is not affected if a Byzantine process leaves: the outlived processes where quorums intersect are by definition a subset of well-behaved processes.

Quorum inclusion. Now let us elaborate on the quorum inclusion property that we just used. When a process p' leaves, it sends *Left* messages to its followers (at either L. 19 or L. 28). The followers later remove p' from their quorums (at L. 29-L. 30). These updates are not atomic and happen over time. Therefore, there might be a window when a process p' is removed from the quorum q_1 (that we saw above), but not yet removed from q_1^*. Therefore, quorum inclusion only eventually holds. However, we observe that in the meanwhile, a weaker notion of quorum inclusion, that we call *active quorum inclusion*, is preserved. It considers inclusion only for the active set of processes $\mathcal{A} = \mathcal{P} \setminus \mathcal{L}$, *i.e.*, it excludes the subset $\mathcal{L}$ of processes that have already left. It requires the quorum q_1^* to be a subset of q_1 modulo $\mathcal{L}$. More precisely, it requires $q_1^* \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_1$. This weaker notion is enough to preserve quorum intersection. In the above discussion for quorum intersection, the process p that remains in the intersection is not in the *tomb* set; therefore, it is an active process. Since it is in q_1^* and q_2^*, by active quorum inclusion, it will be in q_1 and q_2 as well.

References

1 Orestis Alpos, Christian Cachin, and Luca Zanolini. How to trust strangers: Composition of byzantine quorum systems. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 120–131. IEEE, 2021. doi:10.1109/SRDS53918.2021.00021.

2 Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

3 Andrea Bracciali, Davide Grossi, and Ronald de Haan. Decentralization in open quorum systems: Limitative results for ripple and stellar. In *2nd International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2020)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

4 Christian Cachin and Luca Zanolini. From symmetric to asymmetric asynchronous byzantine consensus. *arXiv preprint*, 2020. arXiv:2005.08795.

5 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

6 Ivan Damgård, Yvo Desmedt, Matthias Fitzi, and Jesper Buus Nielsen. Secure protocols with asymmetric trust. In *Advances in Cryptology–ASIACRYPT 2007: 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007. Proceedings 13*, pages 357–375. Springer, 2007. doi:10.1007/978-3-540-76900-2_22.

7 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988. doi:10.1145/42282.42283.

8 Álvaro García-Pérez and Maria A Schett. Deconstructing stellar consensus (extended version). *arXiv preprint*, 2019. arXiv:1911.05145.

9 Andrew Lewis-Pye and Tim Roughgarden. Byzantine generals in the permissionless setting. *arXiv preprint*, 2021. arXiv:2101.07095.

10 Andrew Lewis-Pye and Tim Roughgarden. Permissionless consensus. *arXiv preprint*, 2023. arXiv:2304.14701.

11 Xiao Li, Eric Chan, and Mohsen Lesani. Quorum subsumption for heterogeneous quorum systems. In *37th International Symposium on Distributed Computing (DISC 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

12 Xiao Li and Mohsen Lesani. Reconfigurable heterogeneous quorum systems. arXiv:2304.02156.

13 Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 80–96, 2019.

14 Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

15 David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32:1–45, 2015.

16 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016. doi:10.1145/2976749.2978399.

17 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *White paper*, 2008.

18 David Schwartz, Noah Youngs, and Arthur Britto. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5(8):151, 2014.

19 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022. doi:10.1145/3548606.3559361.

20 Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-bft: High-throughput bft for blockchains. *arXiv preprint*, page 92, 2019. arXiv:1906.05552.

21 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2011.

22 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019. doi:10.1145/3293611.3331591.

Brief Announcement: Concurrent Aggregate Queries

Gal Sela
Technion, Haifa, Israel

Erez Petrank
Technion, Haifa, Israel

Abstract

Concurrent data structures serve as fundamental building blocks for concurrent computing. Many concurrent counterparts have been designed for basic sequential algorithms; however, one notable omission is a concurrent tree that supports aggregate queries. Aggregate queries essentially compile succinct information about a range of data items. Such queries play an essential role in various applications and are commonly taught in undergraduate data structures courses. In this paper, we formalize a type of aggregate queries that can be efficiently supported by concurrent trees and present a design for implementing these queries on concurrent lock-based trees. We present two algorithms implementing this design, where one optimizes for tree update time, while the other optimizes for aggregate query time.

2012 ACM Subject Classification Computing methodologies → Shared memory algorithms; Computing methodologies → Concurrent algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases Concurrent Algorithms, Concurrent Data Structures, Aggregate queries, Range queries, Binary Search Tree, Linearizability

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.53

Related Version *Full Version*: https://arxiv.org/abs/2405.07434 [13]

Funding This work was supported by the Israel Science Foundation Grant No. 1102/21.

1 Introduction

Concurrent programs rely on concurrent data structures as a foundational component. Considerable effort has been dedicated to developing efficient concurrent data structures. However, not all sequential functionalities have been extended to the concurrent setting. In this paper we look at such a functionality whose concurrent version has not been addressed: efficient aggregate queries. An aggregate query is a query whose answer summarizes a range of elements with consecutive keys in the data structure into a succinct value. For instance, a data structure holding employee records sorted by age may be queried regarding the average salary of employees in a certain age range.

It is desirable to build efficient concurrent algorithms for aggregate queries, as sequential aggregate queries are used in various applications, and a concurrent extension may scale their execution on a multi-core machine. For instance, order-statistic trees [2], which support the `select`(i) and `rank`(key) aggregate queries (returning the element with the i-th smallest key, and the position of key, respectively), are used in Python libraries for sorted containers [14, 8] to efficiently support the basic operations of accessing $collection[i]$ and querying $collection.index(key)$ respectively.

Naively, one could answer an aggregate query on a sequential data structure by traversing the relevant elements. The concurrent counterpart would be taking a linearizable snapshot of the data structure and traversing it. Previous works on range queries accomplished that

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 53; pp. 53:1–53:7
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

[15, 11, 10, 1], but the traversal in this approach costs time linear in the number of elements in the queried range, which is highly inefficient for aggregate queries that may be answered using some metadata without traversing all the relevant elements. There has been work on implementing specific concurrent aggregate queries more efficiently: [7] proposed a concurrent array supporting a query that aggregates the values of all cells (and not of an input key range). They addressed only simple static arrays without supporting element insertion or deletion, whereas we address dynamic data sets. [12] proposed a way to efficiently support a `size` query returning the total number of elements in a concurrent set or dictionary. They utilize central metadata regarding the data structure's size to allow aggregate queries to return an answer without accessing the data structure's elements themselves. We, however, aim to efficiently answer queries about an arbitrary key range provided as input to the query. For this, one central metadata that aggregates information about all the keys is insufficient.

We focus our attention on aggregate queries on trees. We look at external binary search trees (where external means they hold the elements in the leaves) though our work could be extended to other trees as well. For efficiently answering aggregate queries on sequential trees, one could place in each tree node suitable metadata that is a function of the elements in the leaves of the node's subtree. For instance, an order-statistic tree augments a tree with a size field expressing the number of elements in the node's subtree. The metadata function should be chosen to be one that effectual operations (we call an operation that modifies the data structure, like a successful `insert` or `delete`, an *effectual operation*) could maintain during their root-to-target-leaf traversal for not harming their asymptotic time complexity, and also one that aggregate queries could use to get an answer via root-to-leaf traversals (instead of naively traversing the relevant elements), thus executing in time linear in the traversed path length instead of at the number of elements in the query's range. We formalize the addressed aggregate functions and queries in Section 2.

The challenge in designing a linearizable [6] extension for such augmented trees stems from the fact that each effectual operation affects multiple locations (its target-leaf area and metadata fields in the nodes along its root-to-leaf path), and yet aggregate queries should obtain a consistent view of the parts they traverse in the data structure, as if each concurrent effectual operation has completely taken place or did not start at all. We employ two mechanisms to achieve that: multi-versioning, and announcements of ongoing effectual operations. The first enables queries to ignore effectual operations that are considered to occur after them, and the second enables them to take into account all effects of effectual operations that are considered to occur before them. These enable support for non-aggregate range queries as well. We demonstrate our design on the lock-based binary search tree of [3, 4], and present two algorithms that implement this design, highlighting the trade-offs between query time and update time. In the full version of the paper [13], we present all details of our design and algorithms, along with a correctness and complexity analysis.

Two independent concurrent works propose other solutions to the problem of concurrent aggregate queries. [9] suggests that all operations add themselves to a queue of operations in each tree node they traverse, and then help all preceding operations in the queue to advance to the appropriate child before proceeding to the next node in their own traversal. This way, a query sees a virtual snapshot of the tree, as if all previous operations have already been executed and none of the subsequent operations have. [9] do not support a failure option for the `insert` and `delete` operations, which may require traversing the tree twice, posing additional challenges. They also optimize only for aggregate query performance while our approach also includes an algorithm that optimizes the performance of the original tree operations. They offer an improved space complexity over the current work as they do not employ multi-versioning, but multi-versioning does come with substantial benefits. It allows to reduce contention, as demonstrated by our `contains` operation that does not help concurrent operations throughout its traversal and by concurrent aggregate queries that do not help each other. Multi-versioning also enables to support aggregate queries that require several serial traversals (e.g., for querying the median key of a given input key range).

Another concurrent work [5] supports aggregate queries using an alternative multi-versioning mechanism. Every insert and delete operation builds from bottom up a tree of immutable version objects mirroring the tree itself, and links each version object to the corresponding tree node. Queries obtain a tree of versions from the root's version pointer, and operate on this tree. Running queries on an immutable copy of the tree allows for simpler concurrency. However, it comes with the cost of excessive allocations: each modification of a leaf by an insert operation or a delete operation requires creating new version objects for all ancestors of that leaf.

2 Aggregate metadata and aggregate queries

We look at aggregate queries on binary trees, using metadata placed in each node aggregating information about its subtree. The basic idea is to use this metadata to answer queries efficiently without traversing all the elements in the query's range, while ensuring that the asymptotic time complexity of insertions and deletions is not substantially harmed. These operations should be able to maintain the metadata during their root-to-leaf traversal, as the only affected metadata should lie along their path to the target key.

The aggregate metadata we will add to tree nodes is the value of an aggregate function f applied to the set of $(key, value)$ elements in the leaves of the node's subtree. An aggregate function is a function $f : \mathcal{P}(A) \setminus \{\phi\} \to B$, where A, B are non-empty sets. Our aggregate functions' domain would be all the non-empty subsets of the set of possible $(key, value)$ elements in the tree's leaves (denoted by A), and the nodes' metadata type is denoted by B. Next we present a definition that will provide a useful property for aggregate functions:

▶ **Definition 1** (additive aggregate function). *An aggregate function $f : \mathcal{P}(A) \setminus \{\phi\} \to B$ is* additive *if there exists a binary operation $\oplus_f : B \times B \to B$ such that $(B, \oplus_f)$ is a commutative semigroup (namely, $\oplus_f$ is associative and commutative) and for every $X \in \mathcal{P}(A) \setminus \{\phi\}$, $f(X) = \oplus_{f\,a \in X} f(\{a\})$.*

We require the metadata in tree nodes to be a value of an additive aggregate function over the set of $(key, value)$ elements in the leaves of the node's subtree. This ensures that the metadata in each node may be directly updated upon an insertion of a $(key, value)$ element to its subtree, to be $old \oplus_f f(\{(key, value)\})$ where old is the old metadata value. To be able to similarly update the nodes' metadata during a root-to-leaf traversal upon deletion, we require that the metadata function satisfy the following property:

▶ **Definition 2** (subtractive aggregate function). *We say that an aggregate function $f : \mathcal{P}(A) \setminus \{\phi\} \to B$ is* subtractive *if it is additive, and there exists a subtractive binary operation $\ominus_f : B \times B \to B$ such that for every disjoint $X_1, X_2 \in \mathcal{P}(A) \setminus \{\phi\}$, $f(X_2) = f(X_1 \cup X_2) \ominus_f f(X_1)$.*

Being able to directly update the metadata in a certain node to reflect a deletion in its subtree, without re-calculating the metadata node by node from the deleted leaf upwards, is required by our algorithms. This is because not only the deletion initiator needs to calculate its effect on the metadata in the deleted leaf's ancestors. Concurrent operations linearized after the deletion might need to do so as well in ancestors mutual with this deletion, and they should not traverse all the way from the deleted leaf to the relevant ancestor which might be costly.

An aggregate query on an augmented tree returns a result based on multiple data elements of the data structure by executing merely root-to-leaf traversals. Metadata obtained during the traversals may be used both to navigate through the tree and to calculate the query's result. Some queries require multiple root-to-leaf traversals for computing their answer. These traversals may be independent of each other, which means they could be executed concurrently, followed by a central calculation of the query's answer using their results. But there are also queries that require a serial execution of traversals, which is the case when each traversal depends on the result of the previous traversal. Accordingly, we define a *simple* aggregate query, which executes only independent traversals. In contrast, a general aggregate query is a chain of one or more simple aggregate queries composed with one another: the user's input is the input to the first simple query in the chain, the output of the i-th simple query in the chain is the input to the $(i+1)$-st query, and the output of the last simple query is the output of the whole query.

A simple aggregate query performs one or more independent root-to-leaf traversals to gather the information required to answer the query, and then computes the answer using the traversals' results. The traversals may be executed concurrently as they are independent of each other. All traversals perform a root-to-leaf traversal on the tree as follows. They maintain the prefix sum f using $\oplus_f$ on the set of $(key, value)$ pairs of all leaves found in subtrees that the traversal has jumped over so far (namely, descended to the right while they were in the left subtree). For each traversed node, this value and the aggregate value of the current left subtree are added using $\oplus_f$ to yield the prefix sum on all leaves with keys smaller than the current key. The computation of this value is made possible using one simple $\oplus_f$ operation thanks to using an additive aggregate function on the subtree's leaves as the node's metadata. Then a query-specific method `shouldDescendRight` is called to determine to which child the traversal should proceed. It takes as inputs the prefix sum up to the current key and the current node's key. In case of descending to the right, the prefix sum is updated to include the leaves of the current left subtree. The traversal stops when it reaches a leaf node, and returns this leaf and the prefix sum which is now the value of the metadata subtractive aggregate function on the set of $(key, value)$ pairs of all leaves with keys smaller than the reached key. A simple aggregate query is defined by as many `shouldDescendRight` methods as the traversals it needs and a method that takes the list of the traversals' outputs and computes the query's answer.

3 Our design for concurrent aggregate queries

We look at binary search trees implementing a dictionary, which is a collection of distinct keys with associated values, supporting an insertion of an input key with the associated input value if the key does not exist or else returns a failure; a deletion which deletes an input key and its value if the key exists and returns the value or else returns a failure; and a `contains` operation which returns the input key's value if it exists else returns NOT_FOUND. We look specifically at external trees, i.e., trees whose items are found in the leaves.

To make sure aggregate queries obtain a consistent view of the parts they traverse in the data structure, even though each concurrent effectual operation modifies multiple locations, each effectual operation and each aggregate query obtains a timestamp. We will ensure that every query observes all modifications related to effectual operations with timestamps $\leq$

its timestamp, and does not see modifications related to effectual operations with a greater timestamp. Our algorithms are linearizable, and the linearization order of effectual operations and aggregate queries respects the timestamp order, where aggregate queries are linearized after effectual operations with the same timestamp.

For a query to consider all modifications by concurrent effectual operations with timestamp $\leq$ its timestamp, ongoing effectual operations announce themselves by adding an announcement object with their details (including their timestamp) to a global announcements object. Queries read these announcements to fill in missing details about them by themselves, and form the desired full view of their traversed path.

To prevent effectual operations, which run concurrently with a query and have a greater timestamp than the query's timestamp, from overriding data the query is about to use with new data, we employ versioning in the spirit of the multi-versioning of [15] for modifiable fields in the tree's nodes. Effectual operations leave old versions of the data for concurrent queries to inspect, and write the new values in new versions they create for the relevant fields. More specifically, we use timestamped version lists for both the child pointers and the added aggregate metadata field in the tree nodes. These versioned fields consist of a linked list of values tagged with descending timestamps. Each query grabs a timestamp and then builds its view of the query path by reading object versions tagged with this timestamp (to be precise, with the biggest timestamp that is $\leq$ this timestamp).

We apply our design to the lock-based binary search tree of [3, 4]. Effectual operations (successful `insert` and `delete`) acquire the necessary locks, and then, before applying the operation to the tree, they globally announce themselves including obtaining a timestamp and update affected aggregate metadata in a root-to-leaf traversal. Failing `insert` and `delete` and `contains` operate as in the base algorithm, but then in the end verify that no ongoing operation has already announced itself and logically deleted the node they found / inserted a node with the key they have not found. Aggregate queries grab a timestamp and gather the announced effectual operations, and then traverse the tree based on the aggregate metadata similarly to a sequential aggregate query, but while obtaining versions of child pointers and of aggregate metadata according to the obtained timestamp and announced effectual operations.

4 The two algorithms implementing our design

Different approaches could be taken toward the implementation of our design, specifically the operations announcement and the aggregate metadata representation, optimizing for the time complexity of either effectual operations or aggregate queries. We present two algorithms implementing our design: `FastUpdateTree` optimizes for tree update time, in fact incurring zero additional asymptotic time on the original tree operations when they do not face concurrent effectual operations on the same key. `FastQueryTree` offers a better worst-time complexity for aggregate queries, which is a function of their traversal length and the number of concurrent operations (and does not depend on the number of elements in the queried range as in the naive implementation).

In our design, effectual operations have to perform several additional steps in which they might potentially contend with operations of other threads: globally announce and unannounce themselves and update the metadata fields affected by the operation. To reduce contention on effectual operations, `FastUpdateTree` lets them work mostly on single-writer fields written only by the thread that performs the operation. The object in which threads announce their effectual operations and the aggregate metadata field in each tree node are both arrays with a single-writer cell per thread. This demonstrates a time-space trade-off, as effectual operations can execute faster by paying in more space.

When effectual operations announce themselves in the announcements array in `FastUpdateTree`, they do not order themselves with respect to each other, and there is no variable they serialize on (like obtaining a unique timestamp). Aggregate queries are the ones to grab a timestamp while incrementing a global *Timestamp* field using a fetch-and-increment. Effectual operations only need to obtain a timestamp bigger than the last query's timestamp, for writing their updates of versioned fields in newer versions, not overriding data the query needs. For that, an effectual operation first announces itself with an unset timestamp and then obtains the global timestamp value and sets it in the announcement's timestamp field. It sets it using a `CAS` because a concurrent aggregate query might have already set it: aggregate queries obtain the global timestamp value and `CAS` it into the announcement's timestamp of each effectual operation with an unset timestamp they encounter in their traversal of the announcements array.

As for the aggregate metadata array in each `FastUpdateTree` node, each of its cells is a versioned field (namely, holds a linked list of versions with timestamps) containing metadata regarding operations on the node's subtree by the thread associated with this cell. Aggregate queries can calculate the total aggregate value from the per-thread values using $\oplus_f$ (the aggregate function's binary operation).

`FastQueryTree` on the other hand favors the performance of aggregate queries, hence does not let them gather values from a per-thread metadata array; instead, it allocates a single versioned metadata field in each tree node. To update such a field, an effectual operation needs to know which effectual operations are ordered before it, in order to update the metadata to reflect all relevant operations that occurred so far. To this end, all effectual operations serialize by enqueueing their announcement object to a global queue, with a timestamp greater by 1 than the timestamp of the preceding announcement in the queue. The timestamps induce a total order on all effectual operations. Aggregate queries obtain a timestamp (that determines which effectual operations they take into account) from the end of the announcements queue. Equipped with this timestamp, they know which version of each aggregate metadata field to obtain and which announced effectual operations they should consider.

References

1 Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. In *PPoPP*, 2018. doi:10.1145/3178487.3178489.

2 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

3 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *ASPLOS*, 2015. doi:10.1145/2694344.2694359.

4 Tudor Alexandru David, Rachid Guerraoui, Tong Che, and Vasileios Trigonakis. Designing ASCY-compliant concurrent search data structures. Technical report, EPFL, 2014.

5 Panagiota Fatourou and Eric Ruppert. Lock-free augmented trees. In *DISC*, 2024.

6 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990. doi:10.1145/78969.78972.

7 Prasad Jayanti. f-arrays: Implementation and applications. In *PODC*, 2002. doi:10.1145/571825.571875.

8 Grant Jenks. Python sorted containers, 2019. URL: https://grantjenks.com/docs/sortedcontainers.

9 Ilya Kokorin, Dan Alistarh, and Vitaly Aksenov. Wait-free trees with asymptotically-efficient range queries. In *IPDPS*, 2024. doi:10.1109/IPDPS57955.2024.00023.

10 Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. Bundling linked data structures for linearizable range queries. In *PPoPP*, 2022. doi:10.1145/3503221.3508412.

11 Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *DISC*, 2013. doi:10.1007/978-3-642-41527-2_16.

12 Gal Sela and Erez Petrank. Concurrent size. *PACMPL*, 6(OOPSLA2), 2022. doi:10.1145/3563300.

13 Gal Sela and Erez Petrank. Concurrent aggregate queries. *arXiv preprint*, 2024. doi:10.48550/arXiv.2405.07434.

14 Daniel Stutzbach. blist: an asymptotically faster list-like type for Python, 2010. URL: http://stutzbachenterprises.com/blist.

15 Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *PPoPP*, 2021. doi:10.1145/3437801.3441602.

Brief Announcement: Colorless Tasks and Extension-Based Proofs

Yusong Shi
Department of Computer Science and Technology, Tsinghua University, Beijing, China

Weidong Liu
Department of Computer Science and Technology, Tsinghua University, Beijing, China
Zhongguancun Laboratory, Beijing, China

Abstract

The concept of extension-based proofs models the idea of a valency argument, which is widely used in distributed computing. Extension-based proofs are limited in power: it has been shown that there is no extension-based proof of the impossibility of a wait-free protocol for (n, k)-set agreement among $n > k \geq 2$ processes. There are only a few tasks that have been proven to have no extension-based proof of the impossibility, since the techniques in these works are closely related to the specific task.

We give a necessary and sufficient condition for colorless tasks to have no extension-based proofs of the impossibility of wait-free protocols in the NIIS model. We introduce a general adversarial strategy decoupled from any concrete task specification. In this strategy, some properties of the chromatic subdivision that is widely used in distributed computing are proved.

2012 ACM Subject Classification Theory of computation → Interactive proof systems; Theory of computation → Distributed algorithms; Theory of computation → Distributed computing models; Theory of computation → Problems, reductions and completeness

Keywords and phrases Colorless tasks, Impossibility proofs, Extension-based proof

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.54

Related Version *Extended Version*: `https://arxiv.org/abs/2303.14769` [12]

Acknowledgements We would like to thank Faith Ellen and Shihao Liu for helpful discussions and the anonymous reviewers for their comments.

1 Introduction

One of the most important results in distributed computing, due to Fischer, Lynch, and Paterson [7], is that there is no deterministic protocol that solves the consensus task in the asynchronous message passing system. The key idea of their proof is called a valency argument, which proves the existence of an infinite execution in which no process terminates.

The (n, k)-set agreement task, which is a generalization of the consensus task, was first proposed by Chaudhuri [6]. The (n, k)-set agreement task was independently shown to have no wait-free protocol by Borowsky and Gafni [4], Herlihy and Shavit [8], and Saks and Zaharoglou [11]. Topological techniques were used to prove these results.

In [1], Alistarh, Aspnes, Ellen, Gelashvili and Zhu pointed out the differences between valency arguments and combinatorial or topological techniques. In the proof by Fischer, Lynch and Paterson, an infinite execution can be constructed by extending an initial execution infinitely often. In contrast, in those proofs using combinatorial techniques, the existence of a bad execution is proved, but not explicitly constructed. [1] generalized this type of proof and called it an extension-based proof. An extension-based proof is defined as an interaction between a prover and a protocol that claims to solve a task. The prover tries to find out some errors in the protocol by submitting queries to the protocol. If the prover manages to do so, then the prover wins against the protocol. If there exists a prover that can win

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 54; pp. 54:1–54:6
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

against any protocol that claims to solve a task, we say that this task has an extension-based impossibility proof. The proof of the impossibility of consensus is an example of an extension-based proof. In the same paper, they showed that there are no extension-based proofs for the impossibility of a wait-free protocol for the (n, k)-set agreement in the non-uniform iterated immediate snapshot (NIIS) model. The same result was proved in the non-uniform iterated snapshot (NIS) model in the journal version [2]. Some tasks [3, 10] that are closely related to the set agreement task and 1-dimensional colorless tasks have also been shown to have no extension-based proofs.

Do other tasks also have no extension-based impossibility proofs? One way to generate new results is to find a condition that characterizes the tasks that have extension-based impossibility proofs. A task is specified by a tuple $(\mathcal{I}, \mathcal{O}, \Delta)$. A protocol solves a task $(\mathcal{I}, \mathcal{O}, \Delta)$ if, starting with any input values in $\mathcal{I}$, processes decide on output values in $\mathcal{O}$ after communicating with each other for some steps according to the protocol, respecting the input/output relation Δ. Both $\mathcal{I}$ and $\mathcal{O}$ are closed under containment, since processes are assumed to be faulty and may crash at any time. We can show that a task $(\mathcal{I}, \mathcal{O}, \Delta)$ has no extension-based proofs if we can design an adversarial strategy that can construct an adaptive protocol that wins against any extension-based prover.

In this paper, we focus on a subset of tasks called *colorless tasks*. A colorless task is defined only in terms of input and output values, without process ids. *All* our discussions use the definition and related consequences of tasks rather than those specified for colorless tasks. So why do we talk about colorless tasks while adopting the form of general tasks? Part of our design needs a property (Property 1) of the input/output relation Δ.

▶ **Property 1.** *In any possible execution, if a process is allowed to output a value v, then any other process that has seen a superset of the values seen by this process is also allowed to output the value v.*

This property is intrinsic for colorless tasks.

2 Model

An *immediate snapshot* (IS) object, introduced by Borowsky and Gafni in [5] consists of an array and supports only one type of operation, called a *writeread* operation, where a process with id i writes a value to the i-th cell of the array and returns a snapshot of the array immediately following the write. The writeread operations performed to some IS object by different processes are said to be concurrent if all snapshots occur after all writes to the array are finished.

The NIIS model assumes an unbounded sequence of IS objects $IS_1, IS_2 \cdots$. $(n+1)$ sequential threads of control, called *processes*, $\Pi = \{p_0, p_1 \dots p_n\}$, communicate through IS objects to solve decision tasks. A *protocol* is a distributed program to solve a task. In any execution of a protocol in the NIIS model, each process p_i performs a writeread operation on each IS object starting from IS_1. Initially, p_i's state contains its identifier i and its input value. Each time p_i performs a writeread operation on some IS object IS_j using its current state s_i as argument, and sets its current state s_i to its identifier i and the response of its writeread operation. Then p_i consults a map δ to determine whether it should terminate and output a value. If $\delta(s_i) \neq \perp$, p_i outputs $\delta(s_i)$ and terminates. Otherwise, it continues to access this next IS object. Therefore, each *NIIS protocol* is determined by a decision map δ from a local state to output values or $\perp$.

A *configuration C* consists of the contents of each shared object and the state of each process. However, since each process remembers its entire history and only process p_i can write to the i-th component of each IS object, a configuration is fully determined by the states of processes in this configuration. An initial configuration consists of the input values and process ids of all processes. A process is *active* in a configuration if it has not terminated. A configuration is terminated if all processes have terminated.

A *scheduler* repeatedly chooses a set of processes that are poised to perform writeread operations on the same IS object concurrently. A *schedule* α is an ordered sequence of sets of processes chosen by the scheduler. Let C be a reachable configuration in which all active processes have accessed the same number of IS objects. For any set P of processes, a *P-only 1-round* schedule from C is an ordered partition of processes in P that are active in C. A *P-only r-round* schedule from C is a schedule $\alpha_1\alpha_2\cdots\alpha_r$ such that each α_i is a P-only 1-round schedule from $C\alpha_1\cdots\alpha_{i-1}$. A *full r-round schedule* from C is a P-only r-round schedule from C where $P = \Pi$.

An *(abstract) simplex* is the set of all subsets of some finite set. There is a natural geometric interpretation of an (abstract) simplex. In this paper, we use the two definitions interchangeably. An n-simplex S spanned by a set of affinely independent vertices $\{\vec{v}_0, \ldots \vec{v}_n\}$ is defined to be the set of all points x such that $x = \sum_{i=0}^{n} t_i\vec{v}_i$ where $\sum_{i=0}^{n} t_i = 1$ and $t_i \geq 0$ for all i. Any simplex T spanned by a subset of $\{\vec{v}_0, \ldots \vec{v}_n\}$ is called a *face* of S. An *(abstract) simplicial complex* is a finite collection $\mathcal{K}$ of sets that is closed under subset: for any set $S \in \mathcal{K}$, if $S^{'} \subseteq S$, then $S^{'} \in \mathcal{K}$.

For a task, all possible input or output values can be represented by a simplicial complex, called an *input complex* $\mathcal{I}$ or an *output complex* $\mathcal{O}$. Each vertex $\vec{s}$ of these simplices is labeled with a process id and a value that are denoted by $ids(\vec{v})$ and $vals(\vec{s})$, respectively. The *topological task specification* is defined as a carrier map that carries each simplex S of the input complex to a subcomplex of the output complex.

Like tasks, protocols can be represented in terms of combinatorial topology. The *i-th protocol complex* consists of all simplices, represents configurations that are reachable from some initial configuration by a i-round schedule. The *i-th execution map* is a carrier map that carries each initial configuration to all configurations reached from it in the i-th protocol complex. A *protocol* is represented by $(\mathcal{I}, \mathcal{P}, \Xi)$ and a simplicial map $\delta : \mathcal{P} \rightarrow \mathcal{O}$ where $\mathcal{I}$ is the input complex, $\mathcal{P}$ is the i-th protocol complex, and Ξ is the i-th execution map, for some non-negative integer i. We say that a protocol $(\mathcal{I}, \mathcal{P}, \Xi)$ *solves a task* $(\mathcal{I}, \mathcal{O}, \Delta)$ if $\delta(\Xi(s^k))$ is in $\Delta(s^k)$ for each $s^k \in \mathcal{I}$.

Hoest and Shavit [9] showed that the i-th protocol complex of an NIIS protocol is equal to $\chi^i(\mathcal{I}, \delta)$, where χ is the non-uniform chromatic subdivision constructed from the NIIS protocol. Let U be any simplex in $\mathcal{I}$. A *partial protocol* δ_U with respect to U specifies whether a process should output a value(and which output if so) in each configuration reached from an initial configuration that contains U, by a schedule in which $ids(U)$ is the first set of processes. The i-th *protocol complex of a partial protocol δ_U with respect to U* is defined as follows.

- $\mathbb{F}_0(U)$ is the set of all simplices in $\mathcal{I}$ that contain U.
- $\mathbb{F}_1(U)$ is the subcomplex of $\chi(\mathbb{F}_0(U), \delta_U)$ consisting of all simplices representing configurations reachable via 1-round schedules in which the processes in $ids(U)$ have the input values $vals(U)$ and $ids(U)$ is the first set of processes to take a step.
- For $i \geq 1$, $\mathbb{F}_{i+1}(U) = \chi(\mathbb{F}_i(U), \delta_U)$ consists of all simplices representing configurations reachable via $(i + 1)$-round schedules in which the processes in $ids(U)$ have the input values $vals(U)$ and $ids(U)$ is the first set of processes to take a step.

Similarly, a partial protocol with respect to U can be represented topologically as $(\mathbb{F}_0(U), \mathbb{F}_i(U), \Xi)$. The i-th *execution map* Ξ is a carrier map that carries each initial configuration in $\mathbb{F}_0(U)$ to all configurations reached from it in $\mathbb{F}_i(U)$. We say that a partial protocol δ_U with respect to U satisfies the task specification Δ if $\delta_U(\Xi(s^k))$ is in $\Delta(s^k)$ for each $s^k \in \mathcal{I}$ where $\Xi(s^k)$ is not empty.

3 Motivation and summary

In this section, we give a description of the motivation behind our necessary and sufficient condition for a colorless task defined by $(\mathcal{I}, \mathcal{O}, \Delta)$ to have no restricted extension-based proofs.

The (n, k)-set agreement task is the first task that was shown to have no extension-based impossibility proofs. As shown in [1], given any extension-based prover, the adversary will pretend to have a protocol for the (n, k)-set agreement task during phase 1 of the interaction. But after the prover chooses a schedule at the end of phase 1, the adversary can assign a valid output value to each undefined configuration that the prover can reach in the later phases. In other words, the adversary has a partial protocol compatible with the existing assigned values that satisfies the task specification of the (n, k)-set agreement after phase 1. We divide the adversarial strategy into two parts: In this first part, the adversary adaptively defines a protocol in response to any specific prover's queries during the first r phases. In the second part, the adversary designs a partial protocol after the end of phase r so that the prover is doomed to lose.

If the adversary can prevent the prover from finding a problem in the first r phases and construct a partial protocol after phase r, no matter what queries the prover makes during the first r phases and which configurations the prover has chosen at the end of the first r phases, we say that the adversary can *finalize after phase* r. We can show that the adversary can win against any extension-based prover, if and only if the adversary can finalize after phase r for some positive integer r.

In this paper, we introduce the idea behind our necessary and sufficient condition for finalization after phase 1. Most of the techniques used in the proof for larger values of r are introduced in the proof of this case.

We start with a necessary condition assuming that no queries are submitted by the prover during phase 1: there must exist a partial protocol with respect to any possible simplex $U \in \mathcal{I}$. We use the asynchronous computability theorem to give a topological condition for a task to have a partial protocol with respect to each $U \in \mathcal{I}$.

Then we allow the prover to submit queries in phase 1. In the protocol complex of a partial protocol, the output values of some configurations may already be determined during the interaction of phase 1. For two simplices U_1 and U_2 in $\mathcal{I}$ and each simplex S in $\mathbb{F}_1(U_1) \cap \mathbb{F}_1(U_2)$, we consider the configuration, denoted by $CEN(S)$, reached from S via a schedule that repeats the set of processes $ids(S)$ until all processes in $ids(S)$ terminate. We say that two partial protocols δ_{U_1} and δ_{U_2} are *compatible* if the output values of $CEN(S)$ are the same under δ_{U_1} and δ_{U_2} for each possible S. A set of partial protocols is *compatible* if any two partial protocols are compatible. We show that an enhanced necessary condition for finalization after phase 1 is that the set of partial protocols is compatible.

Then we prove that a task $(\mathcal{I}, \mathcal{O}, \Delta)$ has a set of compatible partial protocols $\{\delta_U | U \in \mathcal{I}\}$ then the adversary can always finalize after phase 1, by showing how the adversary can construct an adaptive protocol to win against any prover using this set of compatible partial protocols which can be assumed to terminate after r_m rounds.

The adversary uses an infinite sequence of complexes $S^0, S^1 ...$ and an integer t (current complex) to represent the adaptive protocol, in which $S^0 = \mathcal{I}$. Our adversary maintains three invariants in the interaction with an extension-based prover. For each $0 \leq r < t$ and each vertex $v \in S^r$, $\delta(v)$ is defined. If v is a vertex in S^t, then $\delta(v)$ is undefined or $\delta(v) \neq \perp$. If a vertex s represents the state of a process in a configuration that the prover has reached, then $\delta(v)$ is defined. The second invariant is about the safety of the adaptive protocol. The output values defined by the adaptive protocol will not violate the task specification. To achieve this, the adversary defines the δ values using the output values obtained from the set of partial protocols. The third invariant is that the active distance between a configuration terminated with output values given by δ_{U_1} and a configuration terminated with output values given by δ_{U_2}, where $U_2 \neq U_1$ is at least 3.

The adversary sets $\delta(v) = \perp$ for each vertex in S^r where $r \leq r_m$. The only question is to decide δ for a vertex in S^r where $r > r_m$. Each terminated vertex has a simplex U of $\mathcal{I}$ as its label, indicating which partial protocol its δ value is from. If v is reached from some n-simplex s^n in $\mathbb{F}_{r_m}(U)$ and has a label U, the adversary can use the value of $\delta_U(v^{'})$ where $s^n \in \mathbb{F}_{r_m}(U)$ and $v^{'}$ is the vertex of s^n with the same process id as v. A problem here is that sometimes the adversary has to terminate v with a different label $U^{'}$ to avoid an infinite execution.

If an n-simplex s^n in $Ch^{r_m}(\mathcal{I})$ is not in $\mathbb{F}_{r_m}(U^{'})$, but shares a simplex s_s with $\mathbb{F}_{r_m}(U^{'})$, we define an n-simplex in $\mathbb{F}_{r_m}(U^{'})$ as the canonical neighbor of s^n with the label $U^{'}$. If v has the label $U^{'} \neq U$, the adversary can use the value of $\delta_{U'}(N(s^n, U^{'}))$ where $s^n \in \mathbb{F}_{r_m}(U)$ and $v^{'}$ is the vertex of s^n with the same process id as v. An implementation of canonical neighbors is provided such that this assignment of output values does not violate the carrier map.

We show that using our adversarial strategy, the prover cannot win in phase 1, which means that the prover has chosen some configuration to end phase 1. Let U be the simplex in $\mathcal{I}$ representing the first set of processes in the schedule from some initial configuration to the chosen configuration and their input values. In the subdivision of each n-simplex $s^n \in \mathbb{F}_{r_m}(U)$, the δ values of terminated vertices are obtained from $\delta_U(s^n)$ or $\delta_{U'}(N(s^n, U^{'}))$. Configurations with different labels are separated according to invariant (3). Although δ_U or $\delta_{U'}$ are two different partial protocols, they have the same output values for some configuration $CEN(S)$ since they are compatible by assumption for some shared simplex $S \in \mathbb{F}_1(U) \cap \mathbb{F}_1(U^{'})$. There is a sequence of output assignments from $\delta_{U'}(\tau)$ to $\delta_U(CEN(S))$ and then to $\delta_U(s^n)$ for some shared simplex S such that only one process changes its output values in two adjacent output assignments, where τ is a $dim(CEN(S))$-dimensional subsimplex of $N(s^n, U^{'})$. The colorless condition is used here since the dimension of $CEN(S)$ is less than $n = dim(s^n)$. The adversary terminates the vertices adjacent to the vertices terminated with the label $U^{'}$ using the output assignments of this sequence until the output values of the outermost layer are in $\delta_U(s^n)$. Finally, the adversary can define the δ value of each remaining undefined vertex using $\delta_U(s^n)$. A partial protocol with respect to U is constructed.

▶ **Theorem 1.** *For a colorless task $(\mathcal{I}, \mathcal{O}, \Delta)$, there exists an adversary that can finalize after the first round to win against any restricted extension-based prover if and only if there exists a compatible set of partial protocols, each of which corresponds to a simplex $U \in \mathcal{I}$.*

References

1 Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili, and Leqi Zhu. Why extension-based proofs fail. *Proceedings of the 51'st Annual ACM Symposium on Theory of Computing (STOC)*, pages 986–996, 2019. doi:10.1145/3313276.3316407.

2 Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili, and Leqi Zhu. Why extension-based proofs fail. *SIAM Journal on Computing*, 52(4):913–944, 2023. doi:10.1137/20M1375851.

3 Dan Alistarh, Faith Ellen, and Joel Rybicki. Wait-free approximate agreement on graphs. In *Structural Information and Communication Complexity: 28th International Colloquium, SIROCCO 2021, Wrocław, Poland, June 28 – July 1, 2021, Proceedings*, pages 87–105, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-79527-6_6.

4 Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 91–100, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/167088.167119.

5 Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 41–51, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/164051.164056.

6 Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, July 1993. doi:10.1006/inco.1993.1043.

7 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.

8 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, November 1999. doi:10.1145/331524.331529.

9 Gunnar Hoest and Nir Shavit. Toward a topological characterization of asynchronous complexity. *SIAM Journal on Computing*, 36(2):457–497, 2006. doi:10.1137/S0097539701397412.

10 Shihao Liu. The Impossibility of Approximate Agreement on a Larger Class of Graphs. In Eshcar Hillel, Roberto Palmieri, and Etienne Rivière, editors, *26th International Conference on Principles of Distributed Systems (OPODIS 2022)*, volume 253 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.OPODIS.2022.22.

11 Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.*, 29(5):1449–1483, March 2000. doi:10.1137/S0097539796307698.

12 Yusong Shi and Weidong Liu. Colorless tasks and extension-based proofs, 2023. arXiv:2303.14769.

Brief Announcement: Self-Stabilizing Graph Exploration by a Single Agent

Yuichi Sudo
Hosei University, Tokyo, Japan

Fukuhito Ooshita
Fukui University of Technology, Fukui, Japan

Sayaka Kamei
Hiroshima University, Hiroshima, Japan

Abstract

In this paper, we present two self-stabilizing algorithms that enable a single (mobile) agent to explore graphs. The agent visits all nodes starting from any configuration, *i.e.,* regardless of the initial state of the agent, the initial states of all nodes, and the initial location of the agent. We evaluate the algorithms using two metrics: cover time, which is the number of moves required to visit all nodes, and memory usage, which includes the storage needed for the state of the agent and the state of each node. The first algorithm is randomized. Given an integer $c = \Omega(n)$, the cover time of this algorithm is optimal, *i.e.,* $O(m)$ in expectation, and the memory requirements for the agent and each node v are $O(\log c)$ and $O(\log(c + \delta_v))$ bits, respectively, where n and m are the numbers of nodes and edges, respectively, and δ_v is the degree of v. The second algorithm is deterministic. It requires an input integer $k \geq \max(D, \delta_{\max})$, where D and $\delta_{\max}$ are the diameter and the maximum degree of the graph, respectively. The cover time of this algorithm is $O(m + nD)$, and it uses $O(\log k)$ bits both for agent memory and each node.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases mobile agents, self-stabilization, graph exploration

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.55

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2010.08929` [16]

Funding *Yuichi Sudo*: JSPS KAKENHI 20H04140 and 20KK0232, and JST FOREST Program JPMJFR226U.
Fukuhito Ooshita: JSPS KAKENHI 22K11903.
Sayaka Kamei: JSPS KAKENHI 23K11059 and 23K28037.

1 Introduction

We focus on the *exploration problem* involving a single mobile entity, referred to as a mobile agent or simply an *agent*, within any undirected, simple, and connected graph $G = (V, E)$. This agent, functioning as a finite state machine, migrates from node to node via edges at each time step. Upon visiting a node, the agent can access and modify the node's local memory, known as a *whiteboard*. The graph is anonymous, *i.e.,* nodes lack unique identifiers. Our objective is to enable the agent to visit every node in the graph in as few steps as possible while minimizing the memory usage of both the agent and the whiteboards. This exploration problem, fundamental in the study of mobile computing entities, has been extensively studied [13, 11, 18, 7, 8, 15]. Exploration algorithms have frequently served as a foundation for solving other fundamental problems such as rendezvous, gathering, dispersion, and gossip sharing.

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 55; pp. 55:1–55:7
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we tackle the exploration problem under a more challenging setting: *self-stabilizing exploration* [13, 8]. We do not presuppose any specific initial global state (or *configuration*) of the network. This means that at the start of the exploration, (i) the agent's location within G is arbitrary, (ii) the agent's state is arbitrary, and (iii) the content of each whiteboard is arbitrary. The agent is required to visit all nodes in G from any potentially inconsistent configuration. Generally, an algorithm is considered self-stabilizing [4] for problem P if it can solve P starting from any configuration. Self-stabilizing algorithms are capable of handling any type of transient faults, such as temporary memory corruption, making their design both practically and theoretically significant.

Generally speaking, several studies tackle a variety of problems involving mobile agents in the self-stabilizing setting [2, 8, 10]. In this setting, the number of agents in the graph is fixed. In our case (*i.e.,* self-stabilizing exploration by a single agent), the number of agents is always exactly one: we do not consider configurations where no agent exists, or where two or more agents are present. Therefore, this setting may be particularly suitable for applications where physical robots operates in a field represented by an undirected graph, and the robots can leave information in some way at each intersection in the field.

One might think that this problem, self-stabilizing exploration by a single agent, fall outside the scope of distributed computing because only a single mobile agent is considered. However, we believe this is not the case, as the information accessible to the single agent is distributed throughout the entire graph. When minimizing agent memory, the agent must manage the necessary information distributed across the whiteboards throughout the graph. This situation often illustrates the trade-off between time complexity and agent memory, which is one of the essential aspects of distributed computing. Moreover, as mentioned earlier, exploration algorithms often serve as a fundamental building block for addressing other problems related to mobile agents. Therefore, our randomized and deterministic algorithms, introduced in this paper, could be used to solve various (more distributed) problems, such as rendezvous, gathering, gossiping, and leader election, in a self-stabilizing manner.

Throughout this paper, we denote the number of nodes, the number of edges, the diameter of a graph by n, m, and D, respectively. We denote the degree of a node v by δ_v, and define $\delta_{\min} = \min_{v \in V} \delta_v$ and $\delta_{\max} = \max_{v \in V} \delta_v$.

1.1 Related Work

If we are allowed to use randomization, we can easily solve the self-stabilizing exploration with a well known strategy called *the simple random walk.* When the agent visits a node $v \in V$, it simply chooses a node as the next destination uniformly at random among $N(v)$, where $N(v)$ is the set of all neighbors of v in G. In other words, it moves to any node $u \in N(v)$ with probability $P_{v,u} = 1/\delta_v$. It is well known that the agent running this simple algorithm visits all nodes in G within $O(\min(mn, mD \log n))$ steps in expectation where $n = |V|$, $m = |E|$, and D is the diameter of G. (See [1, 9].) Since the agent is oblivious (*i.e.,* the agent does not bring any information at a migration between two nodes) and does not use whiteboards, the simple random walk is obviously a self-stabilizing exploration algorithm.

Ikeda, Kubo, and Yamashita [7] improved the cover time (*i.e.,* the number of steps to visit all nodes) of the simple random walk by setting the transition probability as $P'_{v,u} = \delta_u^{-1/2} / \sum_{w \in N(v)} \delta_w^{-1/2}$ for any $u \in N(v)$. They proved that the cover time of this *biased random walk* is $O(n^2 \log n)$ steps in expectation. However, we cannot use this result directly in our setting because the agent must know the degrees of all neighbors of the current node to compute the next destination. We can implement this random walk, for example, as follows: every time the agent visits node v, it first obtains $(\delta_u)_{u \in N(v)}$ by visiting all v's neighbors in $2\delta_v$ steps, and then decides the next destination according to probability $(P'_{v,u})_{u \in N(v)}$, which is now computable with $(\delta_u)_{u \in N(v)}$. However, this implementation increases the cover time by a factor of at least $\delta_{\min}$ and at most $\delta_{\max}$. Whereas $n^2\delta_{\max} \log n > mn$ always holds, $n^2\delta_{\min} \log n < \min(mn, mD \log n)$ may also hold. Thus, we cannot determine which random walk has smaller cover time without detailed analysis. To bound the space complexity, we must know an upper bound Δ on $\delta_{\max}$ to implement this random walk. If the agent stores $(\delta_u)_{u \in N(v)}$ on v's whiteboard, it uses $O(\log \Delta)$ bits in the agent-memory and $O(\delta_v \log \Delta)$ bits in the whiteboard of each node v. If the agent stores $(\delta_u)_{u \in N(v)}$ only on the agent-memory, it uses $O(\Delta \log \Delta)$ bits in the agent-memory.

The algorithm given by Priezzhev, Dhar, Dhar, and Krishnamurthy [13], which is nowadays well known as the *rotor-router*, solves the self-stabilizing exploration deterministically. The agent is oblivious, but it uses only $O(\log \delta_v)$ bits in the whiteboard of each node $v \in V$. The edges $(\{v, u\})_{u \in N(v)}$ are assumed to be locally labeled by $0, 1, \dots, \delta_v - 1$ in a node v. The whiteboard of each node v has one variable $v.\texttt{last} \in \{0, 1, \dots, \delta_v - 1\}$. Every time the agent visits a node v, it increases $v.\texttt{last}$ by one modulo δ_v and moves to the next node via the edge labeled by the updated value of $v.\texttt{last}$. This simple algorithm guarantees that starting from any configuration, the agent visits all nodes within $O(mD)$ steps [18]. Masuzawa and Tixeuil [8] also gave a deterministic self-stabilizing exploration algorithm. This algorithm itself is designed to solve the gossiping problem where two or more agents have to share their given information with each other. However, this algorithm has a mechanism to visit all the nodes starting from any configuration, which can be seen as a self-stabilizing exploration algorithm. The cover time and the space complexity for the whiteboards of this algorithm are asymptotically the same as those of the rotor-router, while it uses a constant space of the agent-memory, unlike oblivious algorithms such as the rotor-router.

In his seminal paper, Reingold [14] proved that given positive integer N, a Universal Exploration Sequence (UXS) with length $\mathit{poly}(N)$ for (possibly non-simple) connected d-regular graphs with a size of at most N can be explicitly constructed in log-space and, hence, in polynomial time. Although we omit the definition of UXS here, from this result, we can immediately derive a self-stabilizing exploration algorithm for arbitrary graphs whose size is at most N, whose cover time is polynomial in N, with memory requirements of $O(\log N)$ bits for the agent and zero for the whiteboards. One might think that Reingold's UXS was designed for regular graphs, thus questioning its applicability to arbitrary graphs. However, this difference is not significant because we can virtually transform any arbitrary graph into a regular graph by adding self-loops (see [17] for details). Later, Ta-shma and Zwick [17] introduced the concept of a Strongly Universal Exploration Sequence (SUXS) and obtained results similar to those of Reingold, which allow us to improve the cover time of the above-mentioned self-stabilizing exploration algorithm from $\mathit{poly}(N)$ to $\mathit{poly}(n)$. Thus, the cover time no longer depends on a given upper bound N but only on the actual size n.

A few self-stabilizing algorithms were given for mobile agents to solve problems other than exploration. Blin, Potop-Butucaru, and Tixeuil [2] studied the self-stabilizing naming and leader election problem. Masuzawa and Tixeuil [8] gave a self-stabilizing gossiping algorithm. Ooshita, Datta, and Masuzawa [10] gave self-stabilizing rendezvous algorithms.

If we assume a specific initial configuration, that is, if we do not require a self-stabilizing solution, the agent can easily visit all nodes within $2m$ steps with a simple depth first traversal (DFT). Panaite and Pelc [11] gave a faster algorithm, whose cover time is $m + 3n$ steps. They assume that the nodes are labeled by the unique identifiers. Their algorithm uses $O(m \log n)$ bits in the agent-memory, while it does not use whiteboards. Sudo, Baba, Nakamura, Ooshita, Kakugawa, and Masuzawa [15] gave another implementation of this

Table 1 Randomized self-stabilizing graph exploration algorithms for a single agent.

	Expected Cover Time	Agent Memory	Memory on node v
Simple Random Walk	$O(\min(mn, mD\log n))$	0	0
Biased Random Walk [7]	$O(n^2\delta_{\max}\log n)$	$O(\log\Delta)$	$O(\delta_v\log\Delta)$
(require $\Delta \geq \delta_{\max}$)		$O(\Delta\log\Delta)$	0
$\mathcal{R}_c$ (require $c \geq 2$)	$O\left(m \cdot \min\left(D, \frac{n}{c}+1, \frac{D}{c}+\log n\right)\right)$	$O(\log c)$	$O(\log(\delta_v + c))$

Table 2 Deterministic self-stabilizing graph exploration algorithms for a single agent.

	Cover Time	Agent Memory	Memory on node v
Rotor-router [13]	$O(mD)$	0	$O(\log\delta_v)$
$\texttt{UXS}_N$ [14] (require $N \geq n$)	polynomial in N	$O(\log N)$	0
$\texttt{SUXS}_N$ [17] (require $N \geq n$)	polynomial in n	$O(\log N)$	0
2-color DFT [8]	$O(mD)$	$O(1)$	$O(\log\delta_v)$
$\mathcal{D}_k$ (require $k \geq \max(D, \delta_{\max})$)	$O(m+nD)$	$O(\log k)$	$O(\log k)$

algorithm: they removed the assumption of the unique identifiers and reduced the space complexity on the agent-memory from $O(m\log n)$ bits to $O(n)$ bits by using $O(n)$ bits in each whiteboard. It is worthwhile to mention that these algorithms [11, 15] guarantee the termination of the agent after exploration is completed, whereas designing a self-stabilizing exploration algorithm with termination is impossible. Self-stabilization and termination contradict each other by definition: if an agent-state that yields termination exists, the agent never completes exploration when starting exploration with this state. If such state does not exist, the agent never terminates the exploration.

In the classical or standard distributed computing model (excluding mobile agents), the self-stabilizing token circulation problem, particularly self-stabilizing depth-first token circulation (DFTC), has been extensively studied [6, 3, 12]. Introduced by Huang and Chen [6], this problem was addressed with a self-stabilizing DFTC algorithm using $O(\log n)$ bits per process, which was later reduced to $O(\log\delta_{\max})$ bits by Datta, Johnen, Petit, and Villain[3]. Petit [12] developed a time-optimal (*i.e.,* $O(n)$-time) self-stabilizing DFTC algorithm that also requires $O(\log n)$ bits per process. However, these algorithms are not directly applicable to self-stabilizing exploration by a single agent because the network models are fundamentally different. In the standard model, n computational processes can communicate with each other via communication links in parallel, whereas in our model, only a single agent computes and updates the states of nodes in the network, potentially requiring more time to solve problems. On the other hand, one of the main challenges for self-stabilizing token circulation in the standard model is maintaining exactly one token starting from any configuration where there maybe no tokens or where two or more tokens may exist. As mentioned above, this challenge does not apply to our model, where there is always a single agent in any configuration. Yet, many techniques from standard distributed computing might be adaptable for mobile agent algorithms. For example, our self-stabilizing exploration algorithms employ the technique of repeatedly recoloring graph nodes to resolve variable inconsistencies, a common approach in the design of self-stabilizing algorithms (see Dolev, Israeli, and Moran [5]).

1.2 Our Contribution

In this paper, we investigate how short a cover time we can achieve in a self-stabilizing setting. One can easily observe that the cover time is lower bounded by $\Omega(m)$: any deterministic algorithm requires $\Omega(m)$ steps and any randomized algorithm requires $\Omega(m)$ steps in expectation before the agent visits all nodes. (For completeness, we will prove this lower bound as Theorem 3) Our goal is to give an algorithm whose cover time is close to this lower bound with as small complexity of agent-memory and whiteboards as possible.

We give two self-stabilizing exploration algorithms $\mathcal{R}_c$ and $\mathcal{D}_k$, where c and k are the design parameters. The cover times and the space complexities of the proposed algorithms and the existing algorithms are summarized in Tables 1 and 2.

Algorithm $\mathcal{R}_c$ is a randomized algorithm, where the agent visits all nodes within $O\left(m \cdot \min\left(D, \frac{n}{c}+1, \frac{D}{c}+\log n\right)\right)$ steps in expectation and uses $O(\log c)$ bits in the agent-memory and $O(\log \delta + \log c)$ bits of the whiteboard of each node with degree δ. Thus, we have trade-off between the cover time and the space complexity. The larger c we use, the smaller cover time we obtain. In particular, the expected cover time is $O(m \log n)$ steps if we set $c = \Omega(D/\log n)$, and it becomes optimal (*i.e.,* $O(m)$ steps) if we set $c = \Omega(n)$. This means that we require the knowledge of $\Omega(n)$ value to make $\mathcal{R}_c$ time-optimal. Fortunately, this assumption can be ignored from a practical point of view: even if c is extremely larger than n, the overhead will be just an additive factor of $\log c$ in the space complexity. Thus, any large $c \in \mathit{poly}(n) \cap \Omega(n)$ is enough to obtain the optimal cover time and the space complexity of $O(\log n)$ bits both in the agent memory and whiteboards. Moreover, irrespective of parameter $c \geq 2$, the cover time is $O(mD)$ steps with probability 1.

Algorithm $\mathcal{D}_k$ is a deterministic algorithm. The cover time of $\mathcal{D}_k$ is $O(m + nD)$ steps, which does not depend on parameter k, while the agent uses $O(\log k)$ bits both for the agent-memory and the whiteboard of each node. Thus, we do not have trade-off between the cover time and the space complexity. However, unlike $\mathcal{R}_c$, we require an upper bound on the diameter and the maximum degree of the graph, that is, $\mathcal{D}_k$ requires $k \geq \max(D, \delta_{\max})$ to solve a self-stabilizing exploration. If $k < \max(D, \delta_{\max})$, the correctness of $\mathcal{D}_k$ is no longer guaranteed. However, the knowledge of an upper bound on $\max(D, \delta_{\max})$ is not a strong assumption because the space complexity increases only logarithmically in k: we can assign any large $\mathit{poly}(n)$ value for k to satisfy $k \geq \max(D, \delta_{\max})$ while keeping the space complexity of the agent-memory and v's whiteboard bounded by $O(\log n)$ bits. For example, consider the case that we set $k = 2^{500}$. Then, $\mathcal{D}_k$ can fail only if $D \geq 2^{500}$, which is too large to consider in practice. This extremely large value for k results in the increase of the memory usage only by 500 bits in the agent and whiteboards.

It remains open if there is a deterministic self-stabilizing exploration algorithm with optimal cover time, *i.e.,* $O(m)$ steps.

2 Preliminaries

Let $G = (V, E, p)$ be a simple, undirected, and connected graph where V is the set of nodes and E is the set of edges. The edges are locally labeled at each node: we have a family of functions $p = (p_v)_{v \in V}$ such that each $p_v : \{\{v, u\} \mid u \in N(v)\} \to \{0, 1, \dots, \delta_v - 1\}$ uniquely assigns a *port number* to every edge incident to node v. Two port numbers $p_u(e)$ and $p_v(e)$ are independent of each other for edge $e = \{u, v\} \in E$.

An *algorithm* is defined as a 3-tuple $\mathcal{P} = (\phi, \mathcal{M}, \mathcal{W})$, where $\mathcal{M}$ is the set of states for the agent, $\mathcal{W} = (\mathcal{W}_k)_{k \in \mathbb{N}}$ is the family such that $\mathcal{W}_k$ is the set of states for each node with degree k, and ϕ is a function that determines how the agent updates its state (*i.e.,* agent memory) and the state of the current node (*i.e., whiteboard*). At each time step, the agent is located at exactly one node $v \in V$, and moves through an edge incident to v. Every node $v \in V$ has a whiteboard $w(v) \in \mathcal{W}_{\delta_v}$, which the agent can access freely when it visits v. The function ϕ is invoked every time the agent visits a node or when the exploration begins. Suppose that the agent with state $s \in \mathcal{M}$ has moved to node v with state $w(v) = x \in \mathcal{W}_{\delta_v}$ from $u \in N(v)$. Let $p_{\mathrm{in}} = p_v(\{u, v\})$. The function ϕ takes 4-tuple $(\delta_v, p_{\mathrm{in}}, s, x)$ as the input and returns 3-tuple $(p_{\mathrm{out}}, s', x')$ as the output. Then, the agent updates its state to s' and $w(v)$ to x', after which it moves via port p_{out}, that is, it moves to v' such that $p_{\mathrm{out}} = p_v(\{v, v'\})$. At the beginning of an execution, we let p_{in} be an arbitrary integer in $\{0, 1, \dots, \delta_v - 1\}$ where v is the node that the agent exists on. Note that if algorithm $\mathcal{P}$ is randomized one, function ϕ returns the probabilistic distributions for $(p_{\mathrm{out}}, s', x')$.

Given a graph $G = (V, E)$, a *configuration* (or a global state) consists of the location of the agent, the state of the agent (including p_{in}), and the states of all the nodes in V. Algorithm $\mathcal{P}$ is a self-stabilizing exploration algorithm for a class $\mathcal{G}$ of graphs if for any graph $G = (V, E, p) \in \mathcal{G}$, the agent running $\mathcal{P}$ on G eventually visits all the nodes in V at least once starting from any configuration. Note that, by the above definition, any self-stabilizing exploration algorithm ensures that the single agent visits every node infinitely often.

We measure the efficiency of algorithm $\mathcal{P}$ by three metrics: *the cover time*, *the agent memory space*, and *the whiteboard memory space*. All the above metrics are evaluated in the worst-case manner with respect to graph G and an initial configuration. The cover time is defined as the number of moves that the agent makes before it visits all nodes. If algorithm $\mathcal{P}$ is a randomized one, the cover time is evaluated in expectation. The memory spaces of the agent and the whiteboard on node v are just defined as $\log_2 |\mathcal{M}|$ and $\log_2 |\mathcal{W}_{\delta_v}|$, respectively.

3 Main Theorems

The main theorems of this paper are listed below. Due to page limitations, we omit the descriptions of algorithms $\mathcal{R}_c$ and $\mathcal{D}_k$, as well as the proofs for these theorems. Please see the arXiv version [16] for the detailed algorithms and proofs.

▶ **Theorem 1.** *Algorithm $\mathcal{R}_c$ is a randomized self-stabilizing exploration algorithm for all simple, undirected, and connected graphs. Irrespective of c, the cover time is $O(mD)$ steps with probability 1. The expected cover time is $O\left(m \cdot \min\left(D, \frac{n}{c} + 1, \frac{D}{c} + \log n\right)\right)$ steps. The agent memory space is $O(\log c)$ and the memory space of each node v is $O(\log c + \log \delta_v)$.*

▶ **Theorem 2.** *Algorithm $\mathcal{D}_k$ is a deterministic self-stabilizing exploration algorithm for all simple, undirected, and connected graphs with a diameter and maximum degree of at most k. The cover time is $O(m + nD)$ steps, regardless of the value of k. The memory requirement is $O(\log k)$ for both the agent and each node.*

▶ **Theorem 3.** *Let $\mathcal{P}$ be any exploration algorithm. For any positive integers n, m such that $n \geq 3$ and $n - 1 \leq m \leq n(n+1)/2$, there exits a simple, undirected, and connected graph $G = (V, E)$ with $|V| = n$ and $|E| = m$ such that the agent running $\mathcal{P}$ on G starting from some node in V requires at least $(m-1)/4$ steps to visit all nodes in V in expectation.*

References

1 Romas Aleliunas, Richard M Karp, Richard J Lipton, Laszlo Lovasz, and Charles Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pages 218–223. IEEE, 1979.

2 Lélia Blin, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. On the self-stabilization of mobile robots in graphs. In *International Conference On Principles Of Distributed Systems*, pages 301–314. Springer, 2007. doi:10.1007/978-3-540-77096-1_22.

3 Ajoy K Datta, Colette Johnen, Franck Petit, and Vincent Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Computing*, 13(4):207–218, 2000. doi:10.1007/PL00008919.

4 Edsger W Dijkstra. Self-stabilization in spite of distributed control. In *Selected writings on computing: a personal perspective*, pages 41–46. Springer, 1982.

5 Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997. doi:10.1109/71.588622.

6 Shing-Tsaan Huang and Nian-Shing Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7(1):61–66, 1993. doi:10.1007/BF02278857.

7 Satoshi Ikeda, Izumi Kubo, and Masafumi Yamashita. The hitting and cover times of random walks on finite graphs using local degree information. *Theoretical Computer Science*, 410(1):94–100, 2009. doi:10.1016/J.TCS.2008.10.020.

8 Toshimitsu Masuzawa and Sébastien Tixeuil. Quiescence of self-stabilizing gossiping among mobile agents in graphs. *Theoretical computer science*, 411(14-15):1567–1582, 2010. doi:10.1016/J.TCS.2010.01.006.

9 Peter Matthews. Covering problems for markov chains. *The Annals of Probability*, 16(3):1215–1228, 1988.

10 Fukuhito Ooshita, Ajoy K Datta, and Toshimitsu Masuzawa. Self-stabilizing rendezvous of synchronous mobile agents in graphs. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 18–32. Springer, 2017. doi:10.1007/978-3-319-69084-1_2.

11 P. Panaite and A. Pelc. Exploring unknown undirected graphs. *Journal of Algorithms*, 33(2):281–295, 1999. doi:10.1006/JAGM.1999.1043.

12 Franck Petit. Fast self-stabilizing depth-first token circulation. In *International Workshop on Self-Stabilizing Systems*, pages 200–215. Springer, 2001. doi:10.1007/3-540-45438-1_14.

13 Vyatcheslav B Priezzhev, Deepak Dhar, Abhishek Dhar, and Supriya Krishnamurthy. Eulerian walkers as a model of self-organized criticality. *Physical Review Letters*, 77(25):5079, 1996.

14 O Reingold. Undirected connectivity in log-space. *Journal of the ACM (JACM)*, 55(4):1–24, 2008. doi:10.1145/1391289.1391291.

15 Yuichi Sudo, Daisuke Baba, Junya Nakamura, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. A single agent exploration in unknown undirected graphs with whiteboards. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 98(10):2117–2128, 2015. doi:10.1587/TRANSFUN.E98.A.2117.

16 Yuichi Sudo, Fukuhito Ooshita, and Sayaka Kamei. Self-stabilizing graph exploration by a single agent, 2020. arXiv:2010.08929.

17 Amnon Ta-Shma and Uri Zwick. Deterministic rendezvous, treasure hunts, and strongly universal exploration sequences. *ACM Transactions on Algorithms (TALG)*, 10(3):1–15, 2014. doi:10.1145/2601068.

18 Vladimir Yanovski, Israel A Wagner, and Alfred M Bruckstein. A distributed ant algorithm for efficiently patrolling a network. *Algorithmica*, 37(3):165–186, 2003. doi:10.1007/S00453-003-1030-9.